	Type	L #	Hits	Search Text	DBs	Time Stamp
1	BRS	L1	1	(lock or locked or locking) with (frame or block) with (cache or cached or caching) with (task or process or procedure or program) with (recently or recent)	USPAT; US-PGPUB; EPO; JPO; IBM_TDB	2003/12/23 12:20
2	BRS	L2	21	(lock or locked or locking) same (frame or block) with (cache or cached or caching) same (task or process or procedure or program) same (recently or recent)	USPAT; US-PGPUB; EPO; JPO; IBM_TDB	2003/12/23 11:14
3	BRS	L3	312	(lock or locked or locking) with most adj3 (recent or recently)	USPAT; US-PGPUB; EPO; JPO; IBM_TDB	2003/12/23 11:15
4	BRS	L4	16	3 with (cache or cached or caching)	USPAT; US-PGPUB; EPO; JPO; IBM_TDB	2003/12/23 11:16

	Type	L #	Hits	Search Text	DBs	Time Stamp
5	BRS	L5	338	(lock or locked or locking) with (cache or cached or caching) with (task or process or procedure or program)	USPAT; US-PGPUB; EPO; JPO; IBM_TDB	2003/12/23 12:21
6	BRS	L6	19	(lock or locked or locking) adj3 (cache or cached or caching) adj3 (task or process or procedure or program)	USPAT; US-PGPUB; EPO; JPO; IBM_TDB	2003/12/23 12:31
7	BRS	L7	108	(lock or locked or locking) adj3 (cache or cached or caching) with (task or process or procedure or program)	USPAT; US-PGPUB; EPO; JPO; IBM_TDB	2003/12/23 12:31

US006560677B1

(12) United States Patent
Bridges et al.

(10) Patent No.: US 6,560,677 B1
(45) Date of Patent: May 6, 2003

(54) **METHODS, CACHE MEMORIES, SYSTEMS AND COMPUTER PROGRAM PRODUCTS FOR STORING TRANSIENT, NORMAL, AND LOCKED ENTRIES IN AN ASSOCIATIVE CACHE MEMORY**

(75) Inventors: **Jeffrey Todd Bridges**, Raleigh, NC (US); **Thomas Andrew Sartorius**, Raleigh, NC (US)

(73) Assignee: **International Business Machines Corporation**, Armonk, NY (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/304,664**

(22) Filed: **May 4, 1999**

(51) **Int. Cl.**[7] **G06F 12/08**

(52) **U.S. Cl.** **711/129**; 711/128; 711/134

(58) **Field of Search** 711/128, 129, 711/134, 118, 133, 136

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,513,367 A		4/1985	Chan et al.	364/200
4,977,498 A		12/1990	Rastegar et al.	364/200
5,025,366 A		6/1991	Baror	364/200
5,367,653 A		11/1994	Coyle et al.	395/400
5,493,667 A		2/1996	Huck et al.	395/452
5,519,846 A	*	5/1996	Swenson	711/130
5,553,262 A	*	9/1996	Ishida et al.	711/123
5,555,393 A		9/1996	Tanaka et al.	395/460
5,584,014 A		12/1996	Nayfeh et al.	395/461
5,627,992 A		5/1997	Baror	395/460
5,732,242 A	*	3/1998	Mowry	711/136
5,809,527 A		9/1998	Cooper et al.	711/133
6,185,657 B1	*	2/2001	Moyer	711/128
6,202,129 B1	*	3/2001	Palanca et al.	711/133
6,047,358 A1	*	4/2001	Jacobs	711/133
6,219,760 B1	*	4/2001	McMinn	711/137
6,223,256 B1	*	4/2001	Gaither	711/134

OTHER PUBLICATIONS

Intel, SA-1100 Microprocessor, Technical Reference Manual, Sep. 1998.

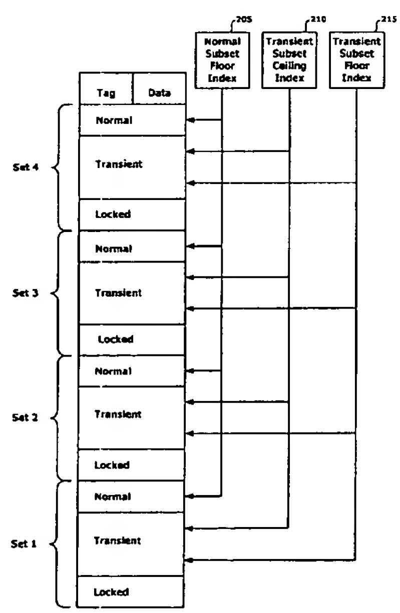
* cited by examiner

Primary Examiner—Gary J Portka
(74) *Attorney, Agent, or Firm*—Myers, Bigel, Sibley & Sajovec, P.A.; John D. Flynn

(57) **ABSTRACT**

Ways of a cache memory system are designated as being in one of three subsets: a normal subset, a transient subset, and a locked subset. The designation of the respective subsets is provided by a normal subset floor index, a transient subset floor index, and a transient subset ceiling index. The respective indexes are used to select the subset into which new entries are copied from main memory as a result of a cache miss. If the new entry is designated as being characterized by normal program behavior, it is copied into the normal subset in the cache. If the new entry is designated as being characterized by transient program behavior, it is copied into the transient subset in the cache. The relationship between the normal subset and the transient subset is programmable. For example, the normal and the transient subsets may include at least one common way of the cache memory or the transient subset may be completely included in the normal subset or completely separate therefrom.

49 Claims, 6 Drawing Sheets



US006044478A

United States Patent [19]

Green

[11] Patent Number: 6,044,478

[45] Date of Patent: Mar. 28, 2000

[54] CACHE WITH FINELY GRANULAR LOCKED-DOWN REGIONS

[75] Inventor: Daniel W. Green, Plano, Tex.

[73] Assignee: National Semiconductor Corporation, Santa Clara, Calif.

[21] Appl. No.: 08/865,909

[22] Filed: May 30, 1997

[51] Int. Cl.^{7} G06F 12/12

[52] U.S. Cl. 714/42; 711/163; 711/141

[58] Field of Search 395/200.33, 375, 395/403, 425, 452; 714/5, 38, 42; 711/163, 141; 709/203

[56] References Cited

U.S. PATENT DOCUMENTS

4,513,367	4/1985	Chan et al. .	
5,230,070	7/1993	Liu	395/425
5,249,286	9/1993	Alpert et al. .	
5,299,318	3/1994	Bernard et al.	395/375
5,353,425	10/1994	Malamy et al. .	
5,390,318	2/1995	Kadangode et al. .	
5,487,162	1/1996	Tetsuya et al. .	
5,493,667	2/1996	Huck et al. .	
5,694,567	12/1997	Bourekas et al.	395/403
5,805,809	9/1998	Singh et al.	395/200.33

OTHER PUBLICATIONS

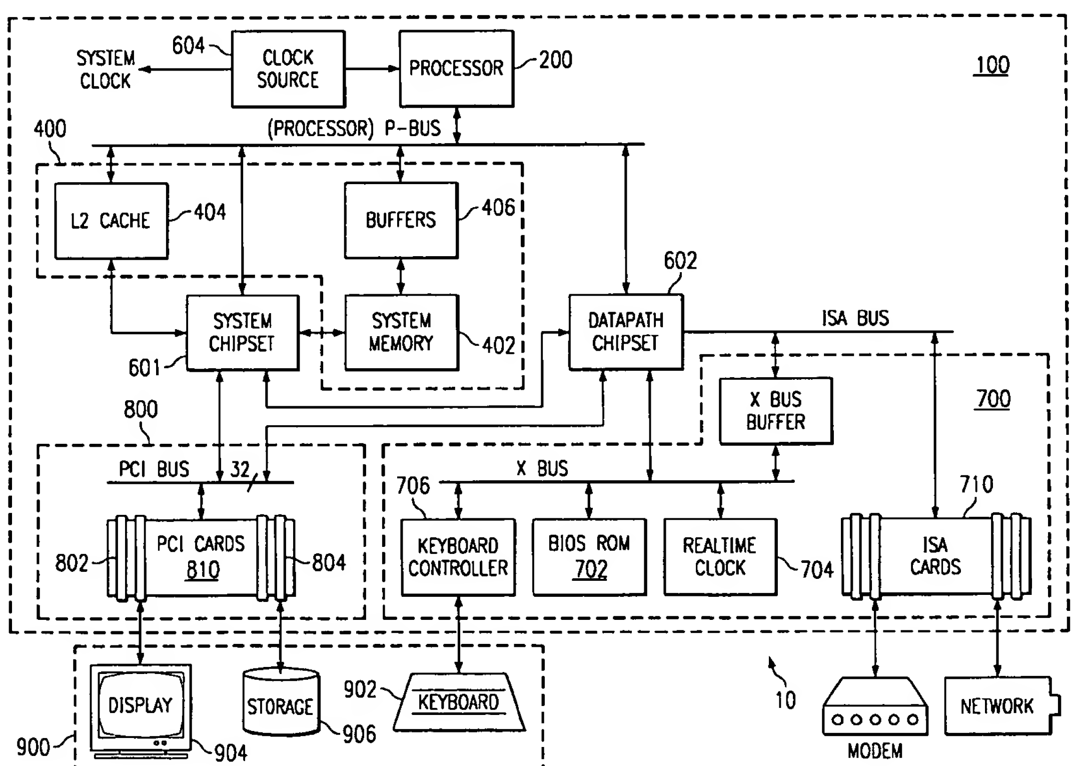
Intel Model Specific Registers and Functions, Chpt. 16, pp. 16–1 through 16–35 Computer Architecture A Quantitative Approach, David A. Patterson, pp. 408–425, Morgan Kaufmann Publishers, Inc. 1990.

Primary Examiner—Joseph E. Palys
Assistant Examiner—Omar A. Omar
Attorney, Agent, or Firm—John L. Maxin

[57] ABSTRACT

A cache has programmable, finely ganular, locked-down regions within a way or way(s) so that the contents of the locked-down regions are not evicted. The finely granular locked-down regions need not be contiguous and are programmed as either "locked-valid" or "locked-invalid" to provide general purpose memory that is local and private to the processor or for masking defected cache lines or portions thereof. Finely granular, programmable spatial regions of the cache that are locked-down are preferably, although not exclusively, programmed through two additional states to the standard MESI (Modified, Exclusive, Shared, Invalid) protocol for multipurpose cache coherency.

15 Claims, 6 Drawing Sheets



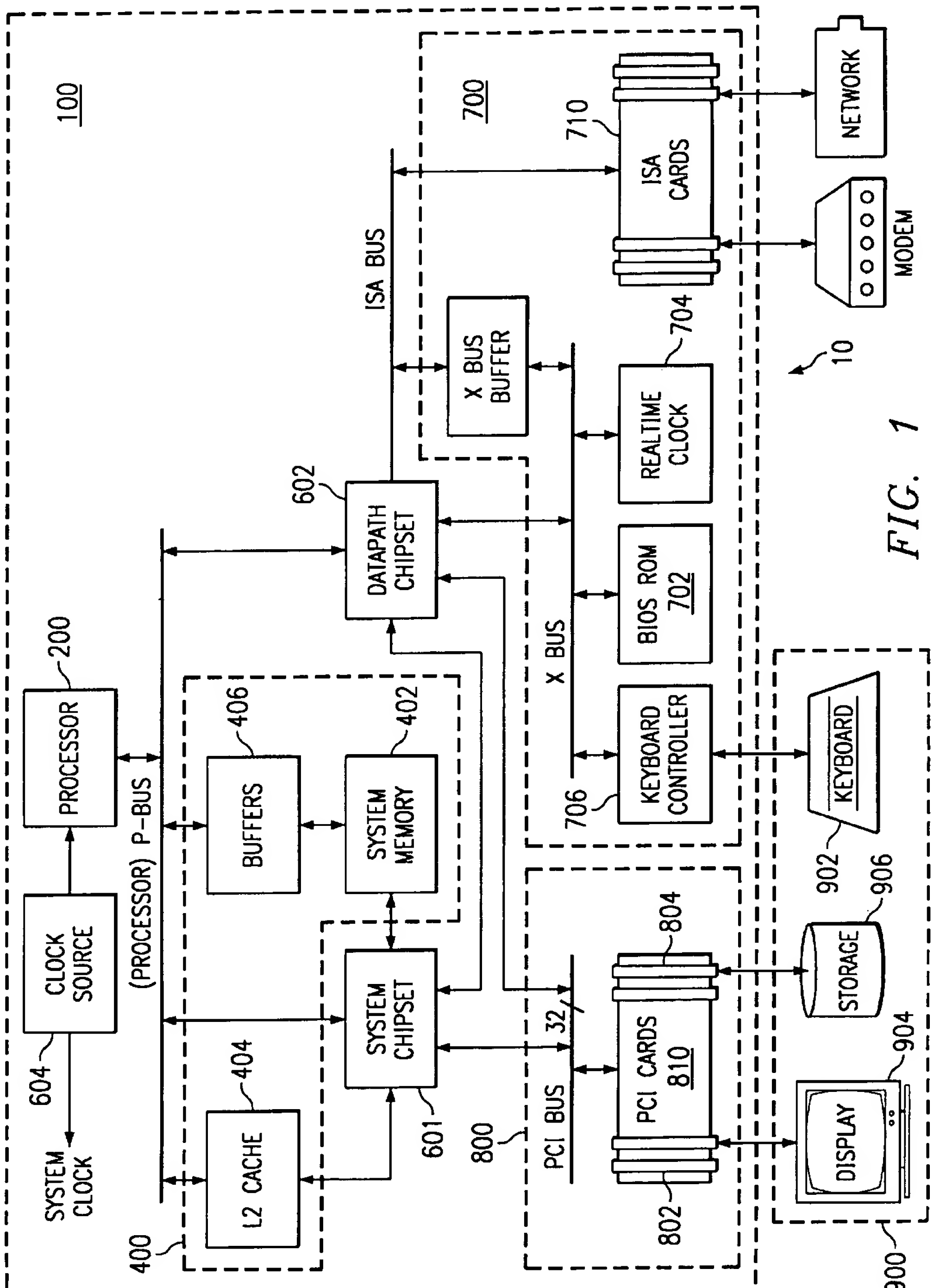

100
700
710
ISA CARDS
NETWORK
MODEM
ISA BUS
X BUS BUFFER
704
REALTIME CLOCK
10
602
DATAPATH CHIPSET
BIOS ROM 702
X BUS
KEYBOARD CONTROLLER
706
KEYBOARD
902
200
PROCESSOR
406
BUFFERS
402
SYSTEM MEMORY
CLOCK SOURCE
(PROCESSOR) P-BUS
SYSTEM CHIPSET
804
32
PCI CARDS 810
906
STORAGE
604
SYSTEM CLOCK
404
L2 CACHE
601
800
PCI BUS
904
DISPLAY
400
802
900

FIG. 1

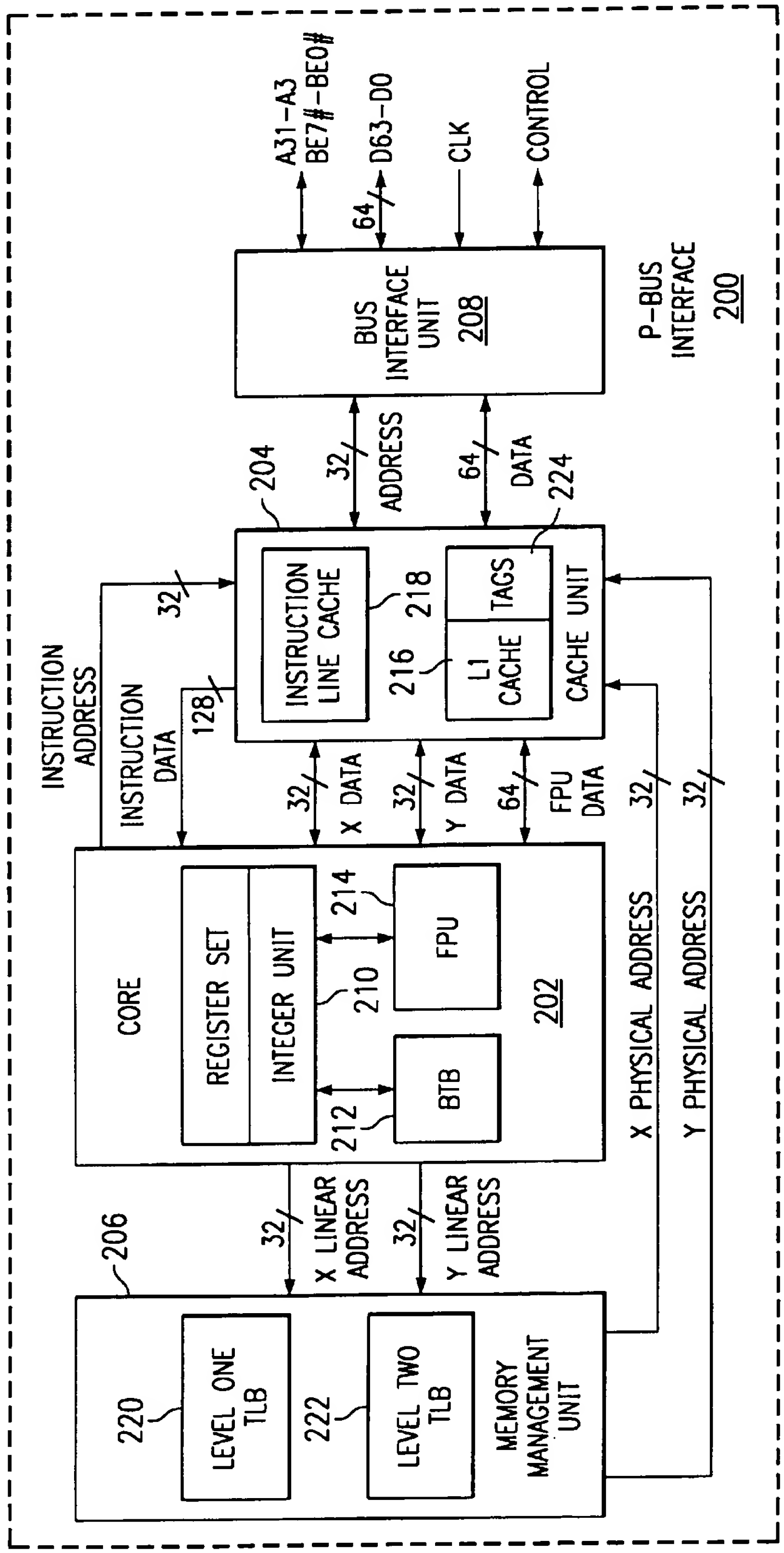

A31-A3
BE7#-BE0#
64
D63-D0
CLK
CONTROL
BUS
INTERFACE
UNIT
208
P-BUS
INTERFACE
200
204
32
ADDRESS
64
DATA
224
INSTRUCTION
ADDRESS
32
INSTRUCTION
LINE CACHE
218
TAGS
INSTRUCTION
DATA
128
216
L1
CACHE
CACHE UNIT
32
X DATA
32
Y DATA
64
FPU
DATA
32
32
CORE
REGISTER SET
INTEGER UNIT
214
FPU
210
202
212
BTB
X PHYSICAL ADDRESS
Y PHYSICAL ADDRESS
206
32
X LINEAR
ADDRESS
32
Y LINEAR
ADDRESS
220
LEVEL ONE
TLB
222
LEVEL TWO
TLB
MEMORY
MANAGEMENT
UNIT

FIG. 2

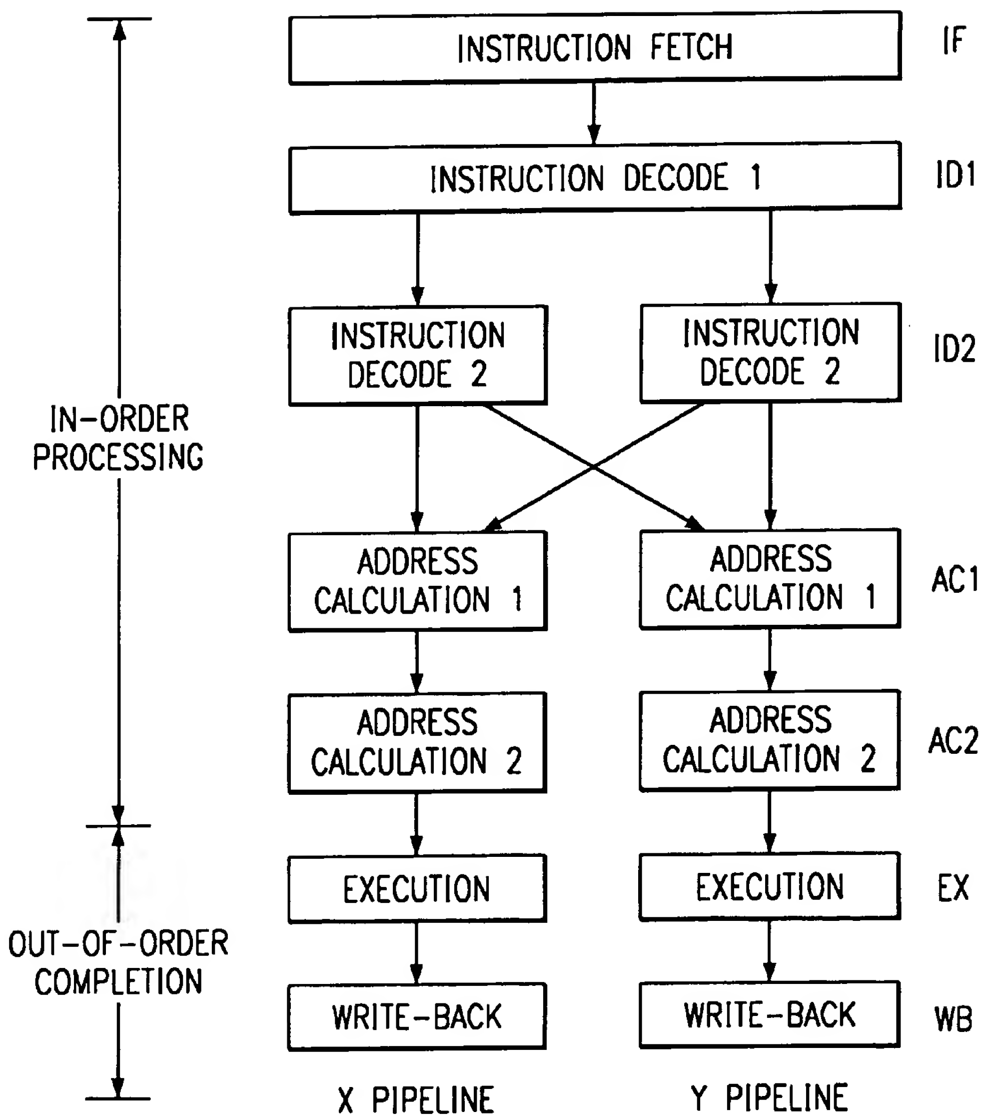
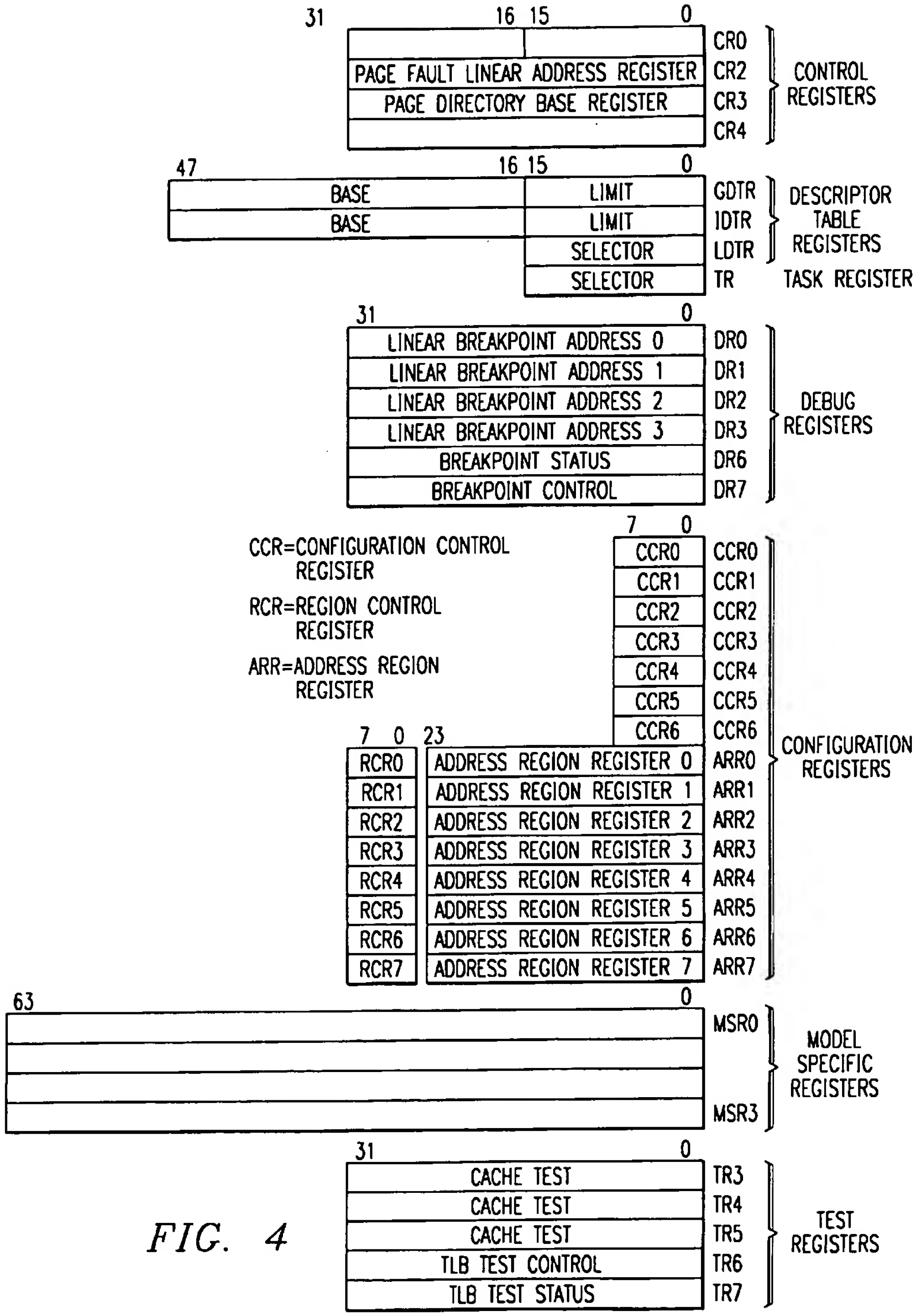
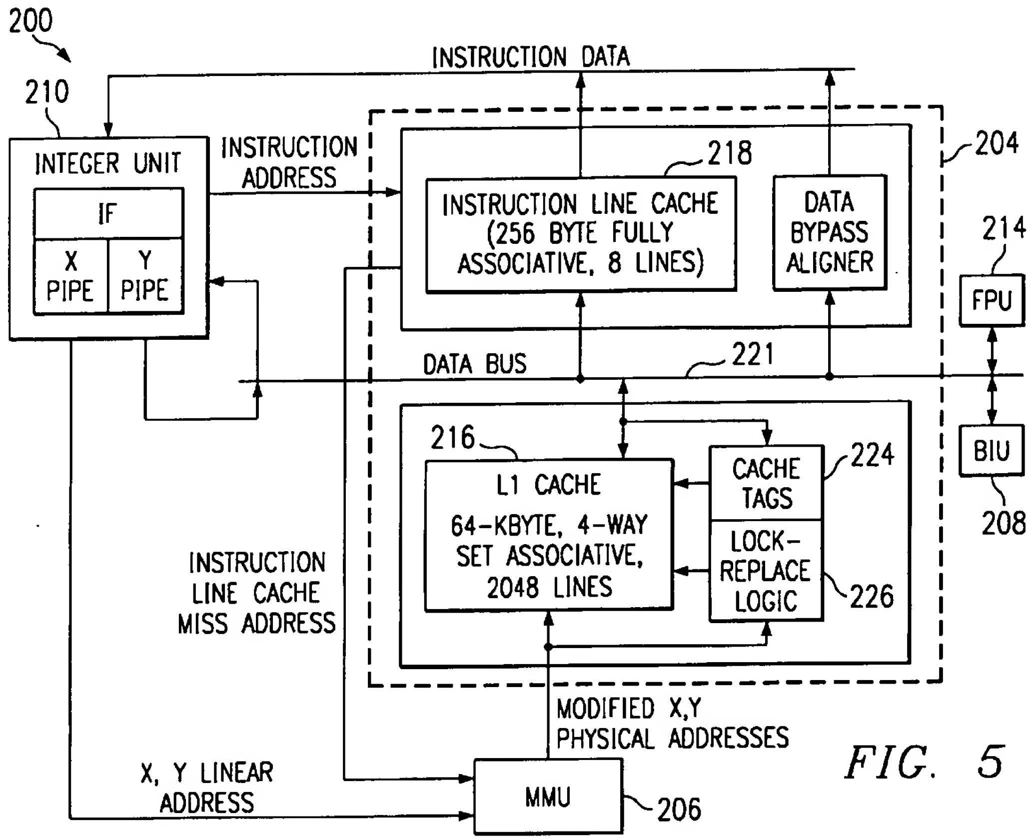

INSTRUCTION FETCH
IF
INSTRUCTION DECODE 1
ID1
INSTRUCTION DECODE 2
INSTRUCTION DECODE 2
ID2
IN-ORDER PROCESSING
ADDRESS CALCULATION 1
ADDRESS CALCULATION 1
AC1
ADDRESS CALCULATION 2
ADDRESS CALCULATION 2
AC2
EXECUTION
EXECUTION
EX
OUT-OF-ORDER COMPLETION
WRITE-BACK
WRITE-BACK
WB
X PIPELINE
Y PIPELINE

FIG. 3



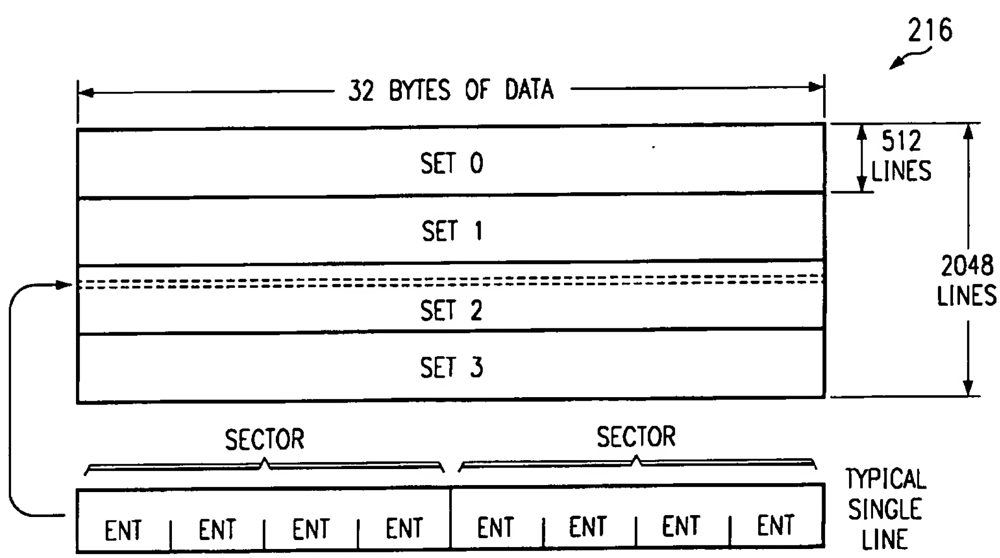
31
16 15
0
CR0
PAGE FAULT LINEAR ADDRESS REGISTER
CR2
PAGE DIRECTORY BASE REGISTER
CR3
CR4
CONTROL REGISTERS
47
BASE
LIMIT
GDTR
BASE
LIMIT
IDTR
SELECTOR
LDTR
SELECTOR
TR
DESCRIPTOR TABLE REGISTERS
TASK REGISTER
LINEAR BREAKPOINT ADDRESS 0
DR0
LINEAR BREAKPOINT ADDRESS 1
DR1
LINEAR BREAKPOINT ADDRESS 2
DR2
LINEAR BREAKPOINT ADDRESS 3
DR3
BREAKPOINT STATUS
DR6
BREAKPOINT CONTROL
DR7
DEBUG REGISTERS
CCR=CONFIGURATION CONTROL REGISTER
RCR=REGION CONTROL REGISTER
ARR=ADDRESS REGION REGISTER
7
CCR0
CCR1
CCR2
CCR3
CCR4
CCR5
CCR6
23
RCR0
ADDRESS REGION REGISTER 0
ARR0
RCR1
ADDRESS REGION REGISTER 1
ARR1
RCR2
ADDRESS REGION REGISTER 2
ARR2
RCR3
ADDRESS REGION REGISTER 3
ARR3
RCR4
ADDRESS REGION REGISTER 4
ARR4
RCR5
ADDRESS REGION REGISTER 5
ARR5
RCR6
ADDRESS REGION REGISTER 6
ARR6
RCR7
ADDRESS REGION REGISTER 7
ARR7
CONFIGURATION REGISTERS
63
MSR0
MSR3
MODEL SPECIFIC REGISTERS
CACHE TEST
TR3
CACHE TEST
TR4
CACHE TEST
TR5
TLB TEST CONTROL
TR6
TLB TEST STATUS
TR7
TEST REGISTERS

FIG. 4



200
INSTRUCTION DATA
210
INTEGER UNIT
IF
X PIPE
Y PIPE
INSTRUCTION ADDRESS
218
INSTRUCTION LINE CACHE (256 BYTE FULLY ASSOCIATIVE, 8 LINES)
DATA BYPASS ALIGNER
204
214
FPU
DATA BUS
221
216
L1 CACHE
64-KBYTE, 4-WAY SET ASSOCIATIVE, 2048 LINES
CACHE TAGS
LOCK-REPLACE LOGIC
224
226
BIU
208
INSTRUCTION LINE CACHE MISS ADDRESS
MODIFIED X,Y PHYSICAL ADDRESSES
X, Y LINEAR ADDRESS
MMU
206

FIG. 5



216
32 BYTES OF DATA
SET 0
SET 1
SET 2
SET 3
512 LINES
2048 LINES
SECTOR
SECTOR
ENT
ENT
ENT
ENT
ENT
ENT
ENT
ENT
TYPICAL SINGLE LINE
ENT = 4 BYTE ENTRY

FIG. 6

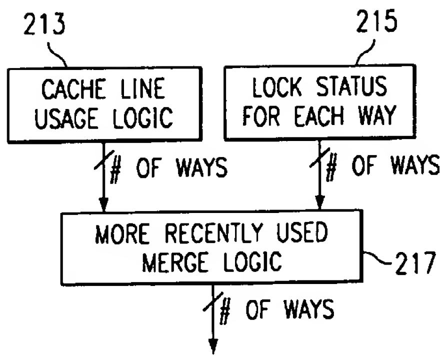


FIG. 7

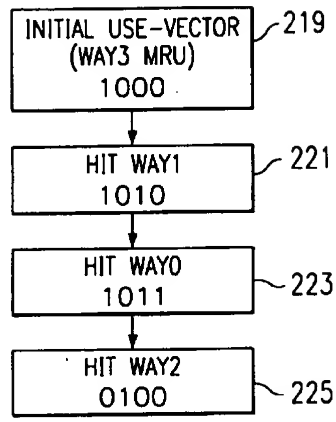


FIG. 8

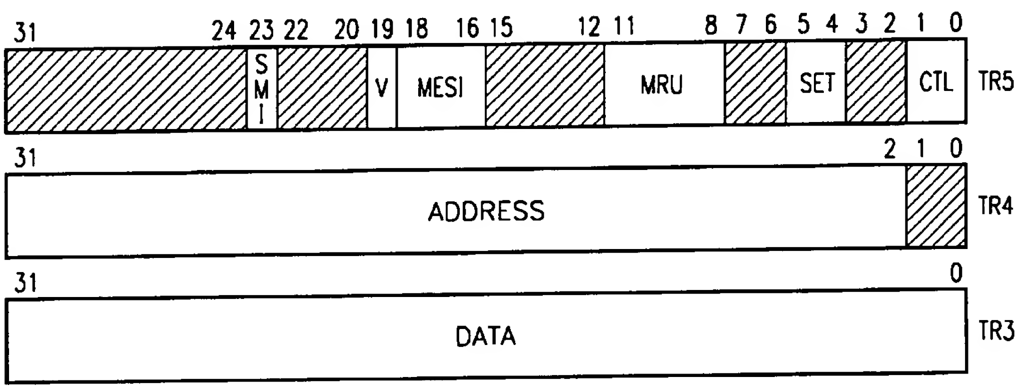


FIG. 9

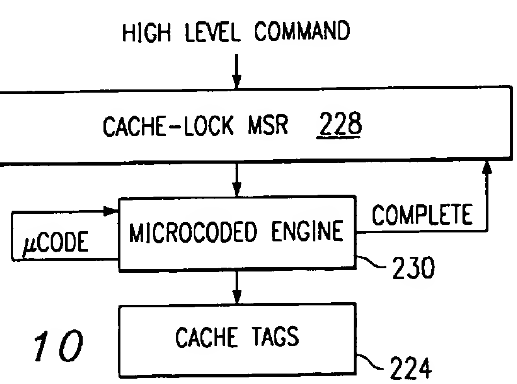


FIG. 10

CACHE WITH FINELY GRANULAR LOCKED-DOWN REGIONS

CROSS-REFERENCE TO RELATED APPLICATION

This patent is related to and commonly assigned U.S. patent application Ser. No. 08/464,921 (Attorney's Docket No. CX-00233), entitled "Partionable Cache Having Spatially Defined Programmable Locked-Down Regions", filed Jun. 5, 1995 and herein incorporated by reference.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The invention relates generally to computer systems, and more specifically to a system and method of caching that permits finely granular, programmable regions to be locked-down thus preventing eviction to provide, among other things, a fast memory scratchpad work area with general application while remaining cache regions provide standard caching functions.

2. Description of Related Art

A cache is a relatively small but fast buffer disposed near the processor for the purpose of reducing latency associated with processor accesses to relatively slow system memory. The cache "shadows" selective portions of the system memory containing temporally or spatially related information acted upon by the processor. Generally speaking, caches can be broadly categorized into two types, namely: a hardware managed array or a software managed array.

The hardware managed type can be generally characterized as an n-way set associative array (where n ranges from direct to fully associative) that replaces entries without any substantial interaction by the operating system or application program software, typically based on the least recently used ("LRU") or most recently used ("MRU") status of the entries. The software managed type typically employs a small random access memory (RAM) that is managed by the operating system or application program software for entry replacement—requiring specific knowledge of the behavior or flow of data or code stored in the cache. Most general purpose processors employ the first type while a large number of digital signal processors (DSPs) employ the second type.

The principal advantage of the first type over the second type is the independence of the cache line replacement policy from the program executed by the processor, particularly useful when executing large application programs or operating systems which tend to jump around to various blocks in address space. A drawback however, of the hardware managed array is that the cache line replacement technique, such as LRU, is not well attuned to certain real-time programming contexts in which, for example, a series of instructions are iteratively executed many times in a time-critical fashion.

The software managed array type of cache is appropriately used by most DSPs, since typically, execution of real-time programming requires program code to stay at or near the top of the memory hierarchy and hence resident in the fastest RAM. DSP-style caches don't suffer from the same "swap-out" problems of the first type cache, but are best applicable to constrained environments where the operating system is relatively simple and the software programs to be executed are known. This restriction arises since software has to explicitly manage the contents of the cache. Thus, the operating system and perhaps even the application

program running on the DSP have to be customized for the particular system and software configuration.

An amalgam of both type of caches can be found, inter alia, in U.S. Pat. No. 5,493,667 to Huck et al. wherein an LRU replacement method and temporal lock-down technique are used to prohibit eviction or invalidation of cache entries in a portion of an instruction cache based on execution of a special "block" instruction. The temporal lock-down technique is intended to hold time critical instruction code (e.g. real-time repetitive/recursive routines) in the instruction cache irrespective of usage status such as LRU. More specifically, a special so-called "block" instruction is executed shortly after an initial execution of the time critical instruction code—freezing the replacement status (e.g. LRU) thus forcing subsequent cache replacements into cache entries other than those occupied by the time critical instruction routine. After the time critical routine is completed, a special so-called "unblock" instruction is executed to release the instruction cache for full general purpose utilization.

A drawback with this approach is the "coarseness" of the defined locked-down regions. For example, if the instruction cache is organized as two way set associative, executing the special block instruction would "block" (freeze) fifty percent of the instruction cache—leaving only the other fifty percent available for general instruction caching functions. Similarly, if the cache is organized as four way set associative, then executing the special block instruction would effectively convert the instruction cache to a three way set associative instruction cache, leaving only seventy five percent of the instruction cache available for general instruction cache purposes—regardless of the size of the time critical routine.

Another drawback with this lock down approach is that it is specific to locking most recent instruction cache entries and devoid of application to locking data in the instruction cache based on the address (as opposed to past temporal use) of that data.

It can be seen therefore, that there is a need for a cache with finely granular locked-down regions suitable for use with general operating system and application programs as well as to effectively accommodate repetitive (recursive) specialized programs.

SUMMARY OF THE INVENTION

To overcome the limitations of the prior art described above, and to overcome other limitations that will become apparent upon reading and understanding the present specification, the present invention discloses a set associative cache having programmable, finely granular locked-down regions within a way or way(s) so that the contents of the locked-down regions are not evicted. The finely granular locked-down regions of the cache need not be contiguous and are programmed as either "Locked-Valid" or "Locked-Invalid" to provide general purpose memory that is local and private to the processor or as a means for masking defective cache lines or portions thereof, respectively. Regions of the cache that are not locked-down provide standard, general purpose, set-associative caching functions. The partitioned cache allows critical data and real-time software programs to remain resident in the locked region while general operating system and application programs use the remainder with a more recently used line replacement technique that can support standard multiprocessor cache coherency protocols.

Finely granular, programmable spatial regions of the cache that are locked-down are preferably, although not

exclusively, programmed with relevant data by executing general purpose load/store type instructions to designated registers within the system register set without the aid of any specialize instructions. A modified cache line replacement technique coined "more recently used" supports the lock-down protocol. In the preferred embodiment, programming the designated registers effectively adds two additional states to the standard MESI (Modified, Exclusive, Shared, Invalid) protocol for multiprocessor cache coherency namely: Locked-Valid and Locked-Invalid. If a location within the cache is identified as Locked-Valid, then a match of its tag will result in a hit and its contents will not be replaced. If a location within the cache is identified as Locked-Invalid, then a match of the tag will result in a miss but the contents will not be replaced. Alternative embodiments may specify the locked-down regions without specific reference to the MESI state or in terms of either physical or logical (virtual) addresses.

A feature of the present invention is that locked-down regions can be programmed in a finely granular fashion without regard to the associativity of the cache.

Another feature of the present invention is that line replacement continues after lock-down using a "more" recently used replacement technique that avoids regions that are locked-down.

Another feature of the present invention is that the unified cache provides programmable, finely granular, locked-down regions to support both instruction and data code.

Another feature of the present invention is that defective cache lines or portions thereof can be masked by the programmable, finely granular, locked-down regions thereby allowing less than perfect caches to be used without allocating substantial portions of the cache as unusable—thus increasing die yields.

Another feature of the present invention is the programmable, finely granular, locked-down regions can be used as a means for storing instruction code executed under the System Management Mode (SMM) of the processor.

Another feature of the present invention is the programmable, finely granular, locked-down regions can be used as a means for time critical-loop program storage such as, for example, a host-based modem or for video/audio compression and/or decompression.

Another feature of the present invention is the programmable, finely granular, locked-down regions can be used as a means for extending the register set of the processor.

Another feature of the present invention is the programmable, finely granular, locked-down regions can be used to store additional or patched code that would extend/repair the instruction set wherein the code may manifest itself as micro-code, RISC (pseudo) op-codes (ROPS), native machine op-codes, or data.

These and various other objects, features, and advantages of novelty which characterize the invention are pointed out with particularity in the claims annexed hereto and forming a part hereof. However, for a better understanding of the invention, its advantages, and the objects obtained by its use, reference should be made to the drawings which form a further part hereof, and to the accompanying descriptive matter, in which there is illustrated and described specific examples of systems and methods employing a cache with finely granular locked-down regions, practiced in accordance with the present invention.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of an exemplary processor system employing a cache practiced in accordance with the principles of the present invention;

FIG. 2 is a block diagram of the processor depicted in FIG. 1;

FIG. 3 is a block diagram of the pipelined stages of the superscalar Integer Unit depicted in FIG. 2;

FIG. 4 is a block diagram of the preferred system register set of the processor depicted in FIG. 2;

FIG. 5 is a block diagram of the processor depicted in FIG. 2 with emphasis on the Cache Unit;

FIG. 6 is a simplified block diagram of the preferred L1 cache portion of the Cache Unit depicted in FIG. 5;

FIG. 7 is a block diagram of modified cache line replacement logic in accordance with the principles of the present invention;

FIG. 8 is a flow diagram of the modified cache line replacement technique;

FIG. 9 is a block diagram of registers associated with configuring the L1 cache depicted in FIG. 6; and,

FIG. 10 depicts an alternative technique for programming locked down regions within the L1 cache.

DESCRIPTION OF THE PREFERRED EMBODIMENT

The detailed description of the preferred embodiment for the present invention is organized as follows:

This organizational table, and the corresponding headings used in this detailed description, are provided for the convenience of reference only and are not intended to limit the scope of the present invention. It is to be understood that while the preferred embodiment is described hereinbelow with respect to the x86 computer architecture, it has generally applicability to any architecture. Certain terminology related to the x86 computer architecture (such as register names, signal nomenclature, etc.) which are known to prac-

titioners in the field of processor design, are not discussed in detail in order not to obscure the disclosure.

Moreover, structural details which will be readily apparent to those skilled in the art having the benefit of the description herein have been illustrated in the drawings by readily understandable block representations and state/flow diagrams, showing and describing details that are pertinent to the present invention. Thus, the illustrations in the figures do not necessarily represent the physical arrangement of the exemplary system, but are primarily intended to illustrate the major structural components in a convenient functional grouping, wherein the present invention may be more readily understood. It is to be understood that other embodiments may be utilized and structural changes may be made without departing from the scope of the present invention.

1. Exemplary Computing System

FIG. **1** illustrates in block diagram form, an exemplary although not exclusive computer system including a system circuit board (a.k.a. motherboard) **100** and various peripherals and peripheral interfaces. Motherboard **100** includes a processor **200** and memory subsystem **400** inter-coupled over a processor P-Bus (sometimes referred to as a processor or local bus). System logic circuitry interfaces the processor **200** to three conventional peripheral buses namely: X-Bus, PCI-Bus, and ISA-Bus. System logic circuitry includes a system chipset **601** and a datapath chipset **602** (sometimes referred to as a North-Bridge and South-Bridge, respectively), as well as an external clock source **604** that provides an external clock input to the processor **200** and a system clock signal to the remainder of the motherboard **100**. The external clock source **604** may take on many forms without departing from the scope of the present invention including a digital or analog phase-locked loop or delay line loop circuitry, the exact details not being necessary for the understanding of the present invention. For the exemplary computer system, the P-Bus is compliant with the so-called "Socket 7" protocol.

Processor **200** and the memory subsystem **400** reside on the P-Bus—the only other direct connections to the P-Bus are the system and datapath chipsets **601** and **602**, respectively. According to the ex emplary division of system logic functions, the system chipset **601** interfaces to a conventional 32-bit PCI-Bus, while the datapath chipset **602** interfaces to the 16-bit ISA-Bus and the internal 8-bit X-Bus. Alternative embodiments allow for a special Advanced Graphics Port (AGP) to interface the P-Bus to a graphics accelerator.

Processor **200** is coupled over the P-Bus to a L2 (level 2) cache **404** and through data buffers **406** to system (DRAM) memory **402**. The system chipset **601** includes control circuitry for the P-Bus, system memory **402**, and the L2 cache **404**. The datapath chipset **602** also interfaces to the conventional X-Bus. The X-Bus is an internal 8-bit bus that couples to the BIOS ROM **702** and the real-time clock (RTC) **704**. In addition, the X-Bus connects to a conventional 8-bit keyboard controller **706**.

The system and datapath chipsets **601** and **602** provide interface control for the 16-bit ISA-Bus and the 32-bit PCI-Bus. The ISA-Bus maintains compatibility with industry standard peripherals, coupling to ISA compliant peripheral card slots **710**. The PCI-Bus provides a higher performance peripheral interface for selected peripherals, including coupling to PCI compliant peripheral card slots **810**—in particular, a video/graphics card **802** provides a video/graphics interface, while a storage controller **804** (which may be included as part of the system chipset **601**) interfaces to mass storage **906**.

The motherboard **100** couples through the PCI, ISA, and X Buses to external peripherals **900**, such as keyboard **902**, display **904**, and mass storage **906**. Network and modem interconnections are provided as ISA cards, but it is to be understood that they could also be provided as PCI cards.

2. Exemplary Processor

Reference is now made to FIG. **2** which depicts a block diagram of the exemplary processor **200** employing a Cache Unit **204** in accordance with the principles of the present invention. It is be understood that other/additional functional partitioning for the processor **200** may be utilized and other modifications can be made, without departing from the scope and spirit of the present invention.

The processor **200** comprises four functional units, namely: i) Core **202**; ii) Cache Unit **204**; iii) Memory Management Unit (MMU) **206**; and, iv) Bus Interface Unit (BIU) **208**.

The Core **202** comprises a superpipelined integer unit (IU) **210**, a Branch Target Buffer (BTB) **212** and a Floating Point Unit (FPU) **214** with provisions to support extensions for multimedia instructions.

The Cache Unit **204** comprises a unified 64K byte L1 cache **216** that stores most recently used data and instruction code and a 256 byte instruction line cache **218** that exclusively stores instruction code.

The MMU **206** translates linear addresses supplied by the IU **210** into a physical addresses for use by the L1 cache **216** and transmission through the BIU **208**. Memory management procedures are preferably x86 compatible, adhering to standard paging mechanisms. The MMU **206** preferably comprises two translation lookaside buffers (TLBs) namely; a main level one (L1) TLB **220** and a larger level two (L2) TLB **222**. The L1 TLB **220** is preferably direct mapped and has sixteen entries to hold forty-two lines. The L2 TLB **222** is preferably six-way set associative and has three hundred eighty four entries to hold three hundred eighty four lines. The Directory Table Entry (DTE) is stored in either the L1 cache **216**, the L2 cache **404**, or in system memory **404**.

The Bus Interface Unit (BIU) **208** provides the P-Bus interface. During a memory cycle, a memory location is selected through address lines (A31–A3 and BE7#–BE0#) on the P-Bus. Data is passed from or to memory through the data lines (D63–D0) on the P-Bus.

The Core **202** receives instructions from the Cache Unit **204**. Integer instructions are decoded by either the X or Y processing pipelines within the superpipelined IU **210**. The instruction is passed to the FPU **214** for processing if the instruction is a floating point or multimedia extension instruction. Data is fetched from the L1 cache **216**. If the data is not in the L1 cache **216**, it is accessed via the BIU **208** from either the L2 cache **404** or system memory **402** (FIG. **1**).

3. The Integer Unit

Reference is now made to FIG. **3** which depicts a block diagram of the superpipelined stages of the IU **210**. Parallel instruction execution is provided with two seven-stage integer pipelines. Each of the two pipelines, X and Y, can process several instructions simultaneously. Each pipeline comprises Instruction Fetch (IF), Instruction Decode 1 (ID1), Instruction Decode 2 (ID2), Address Calculation 1 (AC1), Address Calculation 2 (AC2), Execute (EX), and Write-Back (WB) stages.

The IF stage, shared by both the X and Y pipelines, fetches sixteen bytes of instruction code from the Cache Unit **204** in a single clock cycle. Within the IF stage, the code stream is checked for any branch instructions that could affect normal program sequencing. If an unconditional

or conditional branch is detected, branch prediction logic within the IF stage generates a predicted target address for the instruction. The IF stage then fetches instructions at the predicted address.

The superpipelined Instruction Decode stage comprise sub-stages ID1 and ID2. ID1, shared by both X and Y pipelines, evaluates the code stream provided by the IF stage and determines the number of bytes in each instruction. Preferably at least two instructions per clock are delivered to the ID2 stages, one in each pipeline.

The ID2 stage decodes instructions and sends the decoded instructions to either the X or Y pipeline for execution. The particular pipeline is chosen, based on which instructions are already in each pipeline and how fast they are expected to flow through the remaining pipe-line stages.

The Address Calculation stage comprise sub-stages AC1 and AC2. If the instruction refers to a memory operand, AC1 calculates a linear memory address for the instruction. The AC2 stage performs any required memory management functions, cache accesses, and register file accesses. If a floating point instruction is detected by AC2, the instruction is sent to the FPU **214** for processing. The EX stage executes instructions using the operands provided by the address calculation stage. The WB stage stores execution results either to a register file within the IU **210** or to a write buffer in the Cache Unit **204**.

3.1 Out-of-Order Processing

If an instruction executes faster than the previous instruction in the other pipeline, the instructions may complete out of order. All instructions are processed in order, up to the EX stage. While in the EX and WB stages, instructions may be completed out of order. If there is a data dependency between instructions, hardware interlocks are enforced to ensure correct program execution. Even though instructions may complete out of order, exceptions and writes resulting from the instructions are always issued in program order.

3.2 Pipeline Selection

In most cases, instructions are processed in either X or Y pipeline without pairing constraints on the instructions. However, certain instructions are preferably processed only in the X pipeline namely: branch, floating point, and exclusive instructions. Branch and floating point instructions may be paired with a second instruction in the Y pipeline. Exclusive instructions (e.g. protected mode segment loads, special control, debug and test register accesses, string instructions, multiply and divide, I/O port accesses, push all and pop all, and inter-segment jumps, calls, and returns), that typically require multiple memory accesses, are not preferably paired with instructions in the Y pipeline. Although exclusive instructions are not paired, hardware from both pipelines is used to accelerate instruction completion.

When two instructions that are executing in parallel require access to the same data or register, one of the following types of data dependencies may occur: Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW). Data dependencies typically force serialized execution of instructions. However, the processor **200** employs register renaming, data forwarding, and data bypassing mechanisms that allow parallel execution of instructions containing data dependencies.

3.3 Register Renaming

The processor **200** includes a register file containing 32 physical general purpose registers each of which can be temporarily assigned as one of the general purpose registers defined by the x86 architecture (EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP). For each register write operation, a new physical register is selected to allow previous data to be retained temporarily—effectively removing WAW and WAR dependencies. The programmer does not have to consider register renaming as register renaming is completely transparent to both the operating system and application software.

A WAR dependency exists when the first in a pair of instructions reads a logical register, and the second instruction writes to the same logical register. This type of dependency is illustrated by the pair of instructions shown below:

X PIPE	Y PIPE
(1) MOV BX,AX BX←AX	(2) ADD AX,CX AX←AX+CX

In this and the following examples the original instruction order is shown in parentheses.

In the absence of register renaming, the ADD instruction in the Y pipe would have to be stalled to allow the MOV instruction in the X pipe to read the AX register. The IU **210** however, avoids the Y pipe stall as can be seen with reference to Table 1 below. As each instruction executes, the results are placed in new physical registers to avoid the possibility of overwriting a logical register value and to allow the two instructions to complete in parallel (or out of order) rather than in sequence.

TABLE 1

Register Renaming with WAR Dependency

Instruction	Physical Register Contents					Pipe	Action
	Reg0	Reg1	Reg2	Reg3	Reg4		
(Initial)	AX	BX	CX				
MOV BX,AX	AX		CX	BX		X	Reg3←Reg0
ADD AX,CX			CX	BX	AX	Y	Reg4←Reg0+Reg2

The representation of the MOV and ADD instructions in the final column of Table 1 are completely independent.

A WAW dependency occurs when two consecutive instructions perform writes to the same logical register. This type of dependency is illustrated by the pair of instructions shown below:

X PIPE	Y PIPE
(1) ADD AX,BX AX←AX+BX	(2) MOV AX,[mem] AX←[mem]

Without register renaming, the MOV instruction in the Y pipe would have to be stalled to guarantee that the ADD instruction in the X pipe would first write its results to the AX register. The IU **210** however, avoids the Y pipe stall as can be seen with reference to Table 2 below The contents of the AX and BX registers are placed in physical registers. As each instruction executes, the results are placed in new

physical registers to avoid the possibility of overwriting a logical register value and to allow the two instructions to complete in parallel (or out of order) rather than in sequence.

All subsequent reads of the logical register AX will refer to Reg3, the result of the MOV instruction.

TABLE 2

Register Renaming with WAW Dependency

Instruction	Physical Register Contents Reg0	Reg1	Reg2	Reg3	Pipe	Action
(Initial)	AX	BX				
ADD AX,BX		BX	AX		X	Reg2←Reg0+Reg1
MOV AX, [mem]		BX		AX	Y	Reg3←[mem]

3.4 Data Forwarding

The IU **210** uses two types of data forwarding in conjunction with register renaming to eliminate RAW dependencies namely, operand forwarding and result forwarding.

Operand forwarding takes place when the first in a pair of instructions performs a move from register or memory, and the data that is read by the first instruction is required by the second instruction. The processor performs the read operation and makes the data read available to both instructions simultaneously.

Result forwarding takes place when the first in a pair of instructions performs an operation (such as an ADD) and the result is required by the second instruction to perform a move to a register or memory. The processor **200** performs the required operation and stores the results of the operation to the destination of both instructions simultaneously.

3.4.1 Operand Forwarding

A RAW dependency occurs when the first in a pair of instructions performs a write, and the second instruction reads the same register. This type of dependency is illustrated by the pair of instructions shown below in the X and Y pipelines:

X PIPE	Y PIPE
(1) MOV AX,[mem] AX←[mem]	(2) ADD BX,AX BX←AX+BX

The IU **210** however, avoids the Y pipe stall as can be seen with reference to Table 3 below. Operand forwarding allows simultaneous execution of both instructions by first reading memory and then making the results available to both pipelines in parallel. Operand forwarding can only occur if the first instruction does not modify its source data. In other words, the instruction is a move type instruction (e.g. MOV, POP, LEA). Operand forwarding occurs for both register and memory operands. The size of the first instruction destination and the second instruction source must match.

TABLE 3

Operand Forwarding

Instruction	Physical Register Contents Reg0	Reg1	Reg2	Reg3	Pipe	Action
(Initial)	AX	BX				
MOV AX,[mem]		BX	AX		X	Reg2←Reg2+[mem]
MOV AX,[mem]			AX	BX	Y	Reg3←[mem]+Reg1

3.4.2 Result Forwarding

A RAW dependency can occur when the first in a pair of instructions performs a write, and the second instruction reads the same register. This dependency is illustrated by the pair of instructions in the X and Y pipelines, as shown below:

X PIPE	Y PIPE
(1) ADD AX,BX AX←AX+BX	(2) MOV [mem],AX [mem]←AX

The IU **210** uses result forwarding and avoids the Y pipe stall as can be seen with reference to Table 4 below. Instead of transferring the contents of the AX register to memory, the result of the previous ADD instruction (Reg0+Reg1) is written directly to memory, thereby saving a clock cycle. The second instruction must be a move instruction and the destination of the second instruction may be either a register or memory.

TABLE 4

Result Forwarding

Instruction	Physical Register Contents Reg0	Reg1	Reg2	Pipe	Action
(Initial)	AX	BX			
ADD AX,BX		BX	AX	X	Reg2←Reg0+Reg1
MOV [mem],AX		BX	AX	Y	[mem]←Reg0+Reg1

3.5 Data Bypassing

In addition to register renaming and data forwarding, the IU **210** provides a third data dependency-resolution technique called data bypassing. Data bypassing reduces the performance penalty of those memory data RAW dependencies that cannot be eliminated by data forwarding. Data bypassing is provided when the first in a pair of instructions writes to memory and the second instruction reads the same data from memory. The processor retains the data from the first instruction and passes it to the second instruction, thereby eliminating a memory read cycle. Data bypassing only occurs for cacheable memory locations.

A RAW dependency occurs when the first in a pair of instructions performs a write to memory and the second instruction reads the same memory location. This dependency is illustrated by the pair of instructions in the X and Y pipelines as shown below:

X PIPE	Y PIPE
(1) ADD [mem],AX [mem]←[mem]+AX	(2) SUB BX, [mem] BX←BX-[mem]

The IU **210** uses data bypassing and stalls the Y pipe for only one clock by eliminating the memory read cycle of the Y pipe as can be seen with reference to Table 5 below. Instead of reading memory in the Y pipe, the result of the previous instruction ([mem]+Reg0) is used to subtract from Reg1, thereby saving a memory access cycle.

TABLE 5

Data Bypassing

Instruction	Physical Register Contents Reg0	Reg1	Reg2	Pipe	Action
(Initial)	AX	BX			
ADD [mem],AX	AX	BX		X	[mem]←[mem]+Reg0
SUB BX,[mem]	AX		BX	Y	Reg2←Reg1−{[mem]+Reg0}

3.6 Branch Control

Through simulation and experimentation, the Assignee of the present invention has found that x86 branch instructions occur on average every four to six instructions. The processor **200** minimizes performance degradation and latency of branch instructions through the use of branch prediction and speculative execution. The processor **200** uses a 512-entry, 4-way set associative Branch Target Buffer (BTB) **212** to store branch target addresses and a 1024-entry branch history table. During the IF stage, the instruction stream is checked for the presence of branch instructions. If an unconditional branch instruction is encountered, the processor **200** accesses the BTB **212** to check for the target address of the branch instruction. If the target address is found in the BTB **212**, the processor **200** begins fetching at the target address specified by the BTB **212**.

In the case of conditional branches, the BTB **212** also provides history information to indicate whether the branch is more likely to be taken or not taken. If the conditional branch instruction is found in the BTB **212**, the processor **200** begins fetching instructions at the predicted target address. If the conditional branch misses in the BTB **212**, the processor **200** predicts that the branch will not be taken, and instruction fetching continues with the next sequential instruction. The decision to fetch the taken or not taken target address is preferably, although not necessarily, based on a four-state branch prediction algorithm such as those known within the skill of the art.

Once fetched, a conditional branch instruction is decoded and dispatched to the X pipeline only. The conditional branch instruction proceeds through the X pipeline and is resolved in either the EX stage or the WB stage. The conditional branch is resolved in the EX stage if the instruction responsible for setting the condition codes is completed prior to the execution of the branch. If the instruction that sets the condition codes is executed in parallel with the branch, the conditional branch instruction is resolved in the WB stage.

Correctly predicted branch instructions execute in a single core clock. If resolution of a branch indicates that a misprediction has occurred, the processor **200** flushes the pipeline and starts fetching from the correct target address. The processor **200** preferably pre-fetches both the predicted and the non-predicted path for each conditional branch, thereby eliminating the cache access cycle on a mis-prediction. If the branch is resolved in the EX stage, the resulting misprediction latency is four cycles. If the branch is resolved in the WB stage, the latency is five cycles.

Since the target address of return (RET) instructions is dynamic rather than static, the processor **200** caches target addresses for RET instructions in an eight-entry return stack rather than in the BTB **212**. The return address is pushed on the return stack during a CALL instruction and popped during the corresponding RET instruction.

3.7 Speculative Execution

The processor **200** is capable of speculative execution following a floating point instruction or predicted branch. Speculative execution allows the pipelines to continuously execute instructions following a branch without stalling the pipelines waiting for branch resolution. As will be described hereinbelow, the same mechanism is used to execute floating point instructions in parallel with integer instructions. The processor **200** provides at least four levels of speculation (i.e., combinations of four conditional branches and floating point operations). After generating the fetch address using branch prediction, the processor **200** checkpoints the machine state (registers, flags, and processor environment), increments the speculation level counter, and begins operating on the predicted instruction stream.

Once the branch instruction is resolved, the processor **200** decreases the speculation level. For a correctly predicted branch, the status of the checkpointed resources is cleared. For a branch mis-prediction, the processor **200** generates the correct fetch address and uses the checkpointed values to restore the machine state in a single clock. Writes that result from speculatively executed instructions are not permitted to update the cache or external memory until the appropriate branch is resolved. Speculative execution continues until one of the following conditions occurs: i) a branch or floating point operation is decoded and the speculation level is already at four; ii) an exception or a fault occurs; iii) the write buffers are full; or iv) an attempt is made to modify a non-checkpointed resource (e.g. segment registers and system flags).

3.8 System Register Set

Registers are broadly grouped into the application register set used by application programs and the system register set for use by the operating system program. The application register set preferably includes general purpose registers, segment registers, a flag register, and an instruction pointer register. The system register set includes control registers, system address registers, debug registers, configuration registers, and test registers. In order not to obscure the invention with unessential details, only relevant registers will be described with the understanding that those skilled in the art may easily refer for additional description to publications such as, but not limited to, the Cyrix 6x86*Microprocessor Data Book*, order number: 94175-01, dated March 1996, herein incorporated by reference.

Reference is now made to FIG. 4 which depicts the preferred system register set comprising registers not generally visible to application programmers and typically employed by operating systems and memory management programs. The control registers control certain aspects of the processor **200** such as paging, coprocessor functions, and segment protection. The configuration registers are used, inter alia, to configure the L1 cache **216**, power management features and System Management Mode as well as provide

information on device type and revision. The debug registers provide debugging facilities to enable the use of data access break-points and code execution breakpoints. The test registers provide a mean for programming and testing the contents of both the Cache Unit **204** and the MMU **206**.

The Address Region Registers (ARR0–ARR7) are used to specify the location and size for the eight address regions. Attributes for each address region are specified in the Region Control Registers (RCR0–RCR7). ARR7 and RCR7 are used to define system main memory **402** and differ from ARR0–6 and RCR0–6. The processor **200** eliminates data dependencies and resource conflicts in its execution pipelines by defining non-cacheable regions on-chip. Accesses to address regions defined as non-cacheable by the Region Control Registers are not cached even if the external cache enable pin (KEN#) is active. A register index is used to select one of three bytes in each Address Region Registers. The starting address of the address region, selected by the START ADDRESS field, must be on a block size boundary. For example, a 128K byte block is allowed to have a starting address of 0K bytes, 128K bytes, 256K bytes, and so on.

The Region Control Registers (RCR0–RCR7) specify the attributes associated with the ARRx address regions. Cacheability, weak locking, write gathering, and cache write through policies can be activated or deactivated using the attribute bits defined in the region control registers.

3.8.1 Debug Registers

At least six debug registers (DR0–DR3, DR6 and DR7) support debugging on the processor **200**. Memory addresses loaded in the debug registers, referred to as "breakpoints", generate a debug exception when a memory access of the specified type occurs to the specified address. A data breakpoint can be specified for a particular kind of memory access such as a read or a write. Code breakpoints can also be set allowing debug exceptions to occur whenever a given code access (execution) occurs. The size of the debug target can be preferably set to 1, 2, or 4 bytes. The debug registers are accessed via MOV instructions which can be executed only at privilege level 0. The Debug Address Registers (DR0–DR3) each contain the linear address for one of four possible breakpoints. Each breakpoint is further specified by bits in the Debug Control Register (DR7). For each breakpoint address in DR0–DR3, there are corresponding fields L, R/W, and LEN in DR7 that specify the type of memory access associated with the breakpoint. The R/W field can be used to specify instruction execution as well as data access break-points. Instruction execution breakpoints are preferably taken before execution of the instruction that matches the breakpoint.

The Debug Status Register (DR6) reflects conditions that were in effect at the time the debug exception occurred. The contents of the DR6 register are not automatically cleared by the processor after a debug exception occurs and, therefore, should be cleared by software at the appropriate time. Code execution breakpoints may also be generated by placing the breakpoint instruction (INT 3) at the location where control is to be regained. Additionally, the single-step feature may be enabled by setting the TF flag in the EFLAGS register to cause the processor **200** to perform a debug exception after the execution of every instruction.

3.8.2 Test Registers

As described in more detail hereinbelow, test registers TR3, TR4, and TR5 are used to program/test the L1 cache **216** and test registers TR6 and TR7 are used to program/test the Level One TLB **220**. It is to be understood that the term "test registers" is used merely to generally denote registers within the system register set that are assigned to program/test various subsystems in the processor **200**. With the aid of the present disclosure, those skilled in the art will bring to mind other registers in the system register set to program/test various subsystems without departing from the scope of the present invention.

3.8.3 Model Specific Registers

The processor **200** preferably includes at least four model specific registers (MSRs). MSRs control a variety of hardware and software-related features that are implementation (model) specific such as testability of the cache and/or the BTB or measurement of processor performance. A description of MSRs and the application to the x86 architecture can be found in the *Pentium® Processor Family Developer's Manual*, Order Number 241428-005, (Chapter 16), from Intel Corporation herein incorporated by reference. Generally speaking however, the contents of a particular MSR register specified by the ECX register are preferably read with a so-called "RDMSR" instruction by loading it into the EDX:EAX registers. The contents of a particular MSR register specified by the ECX register are preferably written with a so-called "WRMSR" instruction by loading it into the EDX:EAX registers. Those skilled in the art will bring to mind other forms and implementations of the MSRs without departing from the scope of the present invention.

4. Floating Point Unit

The FPU **214** processes floating point and multimedia extension instructions and is preferably x87 instruction set compatible adhering to the IEEE-754 standard. Floating point instructions may execute in parallel with integer instructions. Integer instructions may complete out of order with respect to the FPU **214** instructions. The processor **200** maintains x86 compatibility by signaling exceptions and issuing write cycles in program order. Floating point instructions are preferably dispatched to the X pipeline in the IU **210**. The address calculation stage of the X pipeline checks for memory management exceptions and accesses memory operands used by the FPU **214**. If no exceptions are detected, the state of the processor **200** is check-pointed and, during AC2, floating point instructions are dispatched to a FPU **214** instruction queue. The processor **200** can then complete subsequent integer instructions speculatively and out of order relative to the floating point instruction and relative to any potential floating point exceptions which may occur.

The processor **200** can preferably dispatch four or more floating point instructions to the floating point instruction queue. The processor **200** continues executing speculatively and out of order, relative to the FPU queue, until one of the conditions that causes speculative execution to halt is encountered. As the FPU **214** completes instructions, the speculation level decreases and the check-pointed resources are available for reuse in subsequent operations. The FPU **214** preferably has a set of six or more write buffers to prevent stalls due to speculative writes.

5. Cache Unit

Reference is now made to FIG. **5** which depicts a block diagram of the processor **200** with emphasis on the exemplary, but not exclusive, Cache Unit **204**. The Cache Unit **204** preferably comprises an L1 cache **216** and an instruction line cache **218**.

The L1 cache **216** is the primary data cache, secondary instruction cache, and is preferably although not exclusively, 64K bytes in size, organized as four-way set-associative, 512 lines per way, with 32 bytes per line (2048 lines) and each line broken into two 16 byte sectors.

The instruction line cache **218** is the primary instruction cache providing a high speed instruction stream to the IU **210** and is preferably although not exclusively, 256 bytes in

size and fully associative. The instruction line cache **218** is filled from the L1 cache **216** over internal bus **221**. Fetches from the IU **210** that hit in the instruction line cache **218** do not access the L1 cache **216**. If an instruction line cache **218** miss occurs, the instruction line data from the L1 cache **216** is transferred simultaneously to the instruction line cache **218** and the IU **210**. The instruction line cache **218** preferably uses a pseudo-LRU replacement algorithm and to ensure proper operation in the case of self-modifying code, any writes to the L1 cache **216** are checked against the contents of the instruction line cache **218**. If a hit occurs in the instruction line cache **218**, the appropriate line is invalidated.

It is to be understood that those skilled in the art will be able to bring to mind other organizations, sizes, associativities, and line replacement techniques for the Cache Unit **204** for which the principles of the present invention may be practiced without departing from the scope of the invention.

6. L1 Cache

Reference is now made to FIG. **6** which depicts more detail of the L1 cache **216**. The L1 cache **216** is preferably divided into four 16K byte sets. Each 16K byte set contains 512 lines. Each line is thirty-two bytes in size and is preferably divided into two 16-byte sectors, each sector described by its own modified MESI bits, described in more detail hereinbelow. When a cache line is allocated, bits A31–A14 of the main memory address are stored in the cache tag **224** (FIG. **5**) with address bits A13–A5 identifying the 32-byte cache line and address bits A4–A2 identifying the specific 4-byte entry within the cache line. The L1 cache **216** is preferably although not exclusively banked and interleaved accessible to allow any two of the following operations to occur in parallel: code fetch, data read (X pipe, Y pipe or FPU) and data write (X pipe, Y pipe or FPU). Cache architectures are ubiquitous and well known in the art. For completeness, however, the book *Computer Architecture A Quantitative Approach*, Section 8.3 pages 408–424, by David A, Patterson and John L. Hennessy, Morgan Kaufmann Publishers, Inc., 1990, a cache architecture primer, is herein incorporated by reference.

7. Replacement/Lock-Down Technique

Reference is now made to FIG. **7** which depicts a block diagram of modified cache line replacement logic in accordance with the principles of the present invention. Cache line usage logic **213** may be any conventional LRU/MRU technique/circuitry and lock status indicator **215** may be any conventional circuitry that indicates whether cache lines, line, or portion thereof are locked. In the preferred embodiment however, the lock status indicator **215** is encoded into the MESI protocol cache coherency state bits which are stored in the cache tag **224** (FIG. **5**). Those skilled in the art will bring to mind with the aid of the present disclosure, other systems and methods for indicating cache line usage and lock status without departing from the scope of the present invention.

As described in more detail hereinbelow, the "more recently used" merge logic **217** inhibits replacement of the contents for any cache line way indicated as being the oldest but also marked as being locked. Rather, the "more recently used" merge logic **217** replaces the cache line contents residing in the next oldest set that is not indicated as being locked. If multiple sets are indicated as being of the same "age", then Way0 is replaced before Way1 which is replaced before Way2 which is replaced before Way3. Those skilled in the art will bring to mind with the aid of the present disclosure, other orders for "tie-breakers" without departing from the scope of the present invention.

Reference is now made to FIG. **8** which depicts a flow diagram of the "more recently used" replacement logic for each cache line/sector in the four Way set associative L1 cache **216**. Each of the four Ways has a bit associated with it indicating that it has been "more" recently used if set (e.g. "1") and "less" recently used if clear (e.g. "0"). Collectively, the four bits form a "Use-Vector". The Use-Vector however, should never be all ones (i.e. "1111") because to do so would not give any indication as to what Way to replace. As described below, if the Use-Vector is about to roll over to all ones ("1111"), then all usage bits except for the use bit the caused the roll over are cleared (i.e. indicate less recently used).

At step **219**, the Use-Vector is depicted as being cleared except for Way3. At step **221**, a hit occurs at Way1 and the Use-Vector becomes 1010. At step **223**, a hit occurs at Way0 and the Use-Vector becomes 1011. At step **225**, a hit occurs at Way2 and the Use-Vector becomes 0100 instead of 1111. This of course assumes that none of the four Ways are locked. If for example, Way3 was locked at step **223**, then the resulting Use-Vector at step **225** would be 1100 instead of 0100.

In the preferred embodiment, the "more" recently used cache line replacement technique can be configured to allocate new lines on read misses only or on read and write misses. Each sector of each cache line is assigned one of six states namely: Unlocked Modified, Unlocked Exclusive, Unlocked Shared, Unlocked-Invalid, Locked-Valid and Locked-Invalid, as defined by V, MESI state bits stored in the cache tag **224** (FIG. **5**). The preferable, although not exclusive approach to programming the modified MESI state bits in the cache tag **224** is through test registers TR3–TR5 existing as part of the system register set described herein above. While the preferred embodiment contemplates assigning a lock status indicator (e.g. V, MESI state bits) for each cache line sector, those skilled in the art will recognize other sizes (e.g. line or multiple lines) for which the lock status indicator may apply and other means for indicating lock status (other than V, MESI), without departing from the scope of the invention.

Cache lines, line, or portions thereof (in the preferred embodiment, "sectors") that have a "Modified" state reflect those that have been updated by the processor **200** but the corresponding location in system memory **402** (FIG. **1**) has not yet been updated by an external write cycle. Modified cache lines/sectors are sometimes referred to as "dirty". Cache lines or portions thereof that have an "Exclusive" state reflect those exclusive to processor **200** and are not duplicated into a cache of another caching agent such as an L1 cache of a second processor in a multiprocessor system (not shown). A write to an exclusive line/sector may be performed without issuing an external write cycle. Cache lines or portions thereof reflect those having a "Shared" state that may be present in the cache of another caching agent within the same system. A write to a shared cache line/sector forces a corresponding external write cycle. Cache lines or portions thereof that have an "Invalid" state do not contain any valid data.

Reference is now made to FIG. **9** which depicts the relevant test registers TR3, TR4, and TR5 within the system register set of the preferred embodiment for programming/testing the L1 cache **216**. It is to be understood that those skilled in the art will bring to mind with the aid of the present disclosure, other means for programming the locked down regions within the L1 cache **216** (such as, but not limited to, MSRs described hereinbelow), without departing from the scope or spirit of the present invention.

For cache writes, test registers TR3 and TR4 are initialized before writing to test register TR5. Eight 4-byte accesses are performed to access a complete cache line. During a cache write, 32-bits of data are written to test register TR3 and tag field information is written to the physical address selected by the ADDRESS field residing in test register TR4 and the SET field in TR5. During a cache read, the physical address resides in the ADDRESS field of test register TR4 and the SET field of test register TR5. The TVB, MESI and MRU fields of test register TR5 are updated with the information from the selected cache line. Test register TR3 returns the selected read data.

A cache flush occurs during a write to test register TR5 if the CTL field is set to zero. During a cache flush, all lines in the L1 cache **216** are read and those with MESI bits indicating "modified" are redefined as "shared". Table 6 below tabularizes the preferred field definitions of test registers TR3, TR4, and TR5.

processor **200** and may be incoherent with other memory in the system. Regions within the L1 cache **216** are locked by setting the V, MESI state bits **(19-16)** of a sector(s) within cache line(s) to "locked-Valid." or "locked-Invalid". Locked regions of the L1 cache **216** are cleared on power-on-reset but are preferably unaffected by a warm reset, assertion of the FLUSH# signal, or execution of the INVD and WBINVD instructions—none of which are described in detail but all of which are within the ordinary skill of the art—particularly with reference to the *Cyrix* 6x86*Processor Data Book*, Order Number 94175-01, dated March 1996, which was herein incorporated by reference.

The L1 cache **216** is preferably flushed before physical addresses are locked down. At least one set within the L1 cache **216** preferably remains unlocked for standard caching operations. Physical addresses to be locked down are first checked against previously locked addresses to avoid aliasing (creating synonyms). Table 7 below tabularizes the cache locking operations.

TABLE 6

Field Definitions of TR3, TR4, and TR5

REGISTER NAME	FIELD NAME	RANGE	DESCRIPTION
TR5	SMI	23	SMIAddressBit Selects separate/cacheable SMI code/data space
	V, MESI	19–16	Valid,MESIBits If = 1000, Modified If = 1001, Shared If = 1010, Exclusive If = 0011, Invalid If = 1100, Locked-Valid If = 0111, Locked-Invalid Else = Undefined
	MRU	11–8	Used to determine the MoreRecently Used set
	SET	5–4	Cache Set Selects one of four cache sets to perform operation on.
	CTL	1–0	Controlfield If = 00: flush cache without invalidate If = 01: write cache If = 10: read cache If = 11: no cache or test register modification
TR4	ADDRESS	31–2	PhysicalAddress
TR3	DATA	31–0	Data written or read during a cache test

The L1 cache **216** can "lock down" regions on a sector by sector basis. These programmable regions are local to the

TABLE 7

Cache Locking Operations

Read/Write	ECX	EDX	EAX	Operation
Read/Write	03h	—	Data to be read or written from/to the cache	Loads or stores data to/from TR3.
Write	04h	—	32 bits of address	Address in EAX is loaded into TR4 to identify which cache line to be locked down.
Read	04h	—	32 bits of address	Stores the contents of TR4 in EAX
Write	05h	—	Data to be written into TR5	Performs operation specified in CTL field of TR5.
Read	05h	—	Data in TR5 register	Reads data in TR5 and stores in EAX

8. Alternative Approach For Programming Locked-Down Regions

Reference is now made to FIG. 10 which depicts an alternative technique for programming locked down regions within the L1 cache **216**. High level "sophisticated" commands are passed through a so-called "cache-lock" model specific register (MSR) **228**. It is to be understood that that cache-lock MSR **228** may be a plurality of registers without departing from the scope of the present invention. The high level command need not reference physical addresses to lock down regions within the L1 cache **216**. Rather, the high level command may pass parameters through the cache-lock MSR **228** indicating its needs (e.g. it needs a 5k byte block of RAM) to drive a microcoded engine **230**. The microcoded engine **230** then sets the state bits in the cache tag **224** to Locked-Valid for a 5K byte block of memory within the L1 cache **216**. A status bit is sent back by the microcoded engine **230** to indicate completion. The status bit may take the form of an exception, an interrupt, or as a settable bit in the cache-lock MSR **228**. The use of high level "sophisticated" commands and the "cache-lock" MSR **228** relieves the programmer of setting individual lock status/valid bits (e.g. MESI state bits) for each line/sector. Exemplary high level commands for the cache-lock MSR **228** may include, but are not limited to:

TSTLOCKABILITY—test if the region sought to be locked down in the L1 cache **216** is in fact, lockable. For example, the region sought to be locked-down may exceed the available space in the L1 cache **216**.

LOCKFILL—Fill a specified locked-down region in the L1 cache **216** with all zeros, ones, or with existing data in the L1 cache **216** without first making it coherent with system memory **402**.

LCOHERENT—Make a specified locked-down region in the L1 cache **216** coherent with system memory **402** before locking.

UNLOCK—Unlock specified regions in the L1 cache **216**.

LOCKINV—Lock-down but invalidate specified locked-down regions in the L1 cache **216**.

With the aid of the present disclosure, those skilled in the art will bring to mind many specific implementations for the microcoded engine **230** and other means and methods for programming/locking regions for cache line/sectors on a finely granular basis without departing from the scope of the present invention.

9. Superscalar Conflict Resolution

In the case of a superscalar architecture (e.g. X and Y pipes), the present invention contemplates a "de-pipelined" cache bus to avoid replacement-lock conflicts between pipes.

10. Exemplary Applications

10.1 Masking Defective Cells In The Cache

A practical application for locking finely granular regions of the L1 cache **216** is avoidance of having to scrap an expensive die. This application is most salient in an unitary die that includes the processor Core **202**, Cache Unit **204**, MMU **206**, and the BIU **208**. Before the present invention, the entire die was scraped if even only one out of thousands of bytes in the L1 cache **216** was defective. In accordance with the present invention, a diagnostic program (which may be part of BIOS or the operating system) is run on power up of the processor **200** to check the integrity of the L1 cache **216**. The diagnostic program sets the MESI protocol as Locked-Invalid for each sector (16 bytes in the preferred embodiment) in which the defective bit or byte resides. Accordingly, the die is usable except that the L1 cache **216** shrinks in size by the sector size (worst case) for each defective byte. The shrinkage of the L1 cache **216** is less however, if multiple defective bytes reside within a single sector.

10.2 SMI Applications

The preferred embodiment supports a system management interrupt (SMI) which invokes a supervisory system management mode (SMM). An exemplary, although not exclusive SMM is disclosed in pending U.S. patent application Ser. No. 08/388,127, filed Mar. 9, 1995, entitled "Enhanced System Management Method And Apparatus With Added Functionality", which is a file-wrapper-continuation of U.S. patent application Ser. No. 08/062,014, which is a continuation-in-part application of U.S. patent application Ser. No. 07/900,052, filed Jun. 17, 1992, all assigned to the Assignee of the present invention, and all herein incorporated by reference.

The so-called SMM "header" is a "stack-like" area in memory which holds the state of the processor **200** before entrance into SMM. If the SMM header resides in system memory **402**, many clock cycles are expended thus a long latency is induced, when SMM is invoked. Accordingly, the present invention locks down (dedicates) cache line sectors in the L1 cache **216** to hold the SMM header thus speeding up entry and exit to SMM.

Moreover until now, program code written for debugging purposes for execution under SMM was not cached due to the potential that it would evict data from the program being debugged and thus impact the debug purpose. SMM code can now be stored in the L1 cache **216** and locked-down to avoid such a problem.

10.3. Critical-Loop Program Storage

Time-critical program loops such as an SMM handler (i.e. the section of the SMM program that resolves the source of the SMI and vectors off to the correct routine) and inner loops of an emulation program (particularly real-time devices such as video/audio compression, modems and audio) are preferably stored in a locked regions in the L1 cache **216** in accordance with the present invention. In this manner, the processor **200** can execute critical loop programs in a zero-wait state fashion.

10.4. Critical Data Storage

The locked down area of the L1 cache **216** can extend the register set of the processor **200** without inducing wait states.

10.5 Micro-Architecture Extension

The locked-down regions of the L1 cache **216** can be used to store additional or "patched" code that would extend/repair the instruction set of the processor **200**. The code may manifest itself as micro-code, RISC (pseudo) op-codes (ROPS), native machine op-codes, or data and would be accessible through an internal trap mechanism such as, but not limited to, a fast entry/exit SMI.

11. Conclusion

Although the Detailed Description of the invention has been directed to certain exemplary embodiments, various modifications of these embodiments, as well as alternative embodiments, including extensions to other levels of cache (e.g. L2 cache) will be suggested to those skilled in the art. The invention encompasses any modifications or alternative embodiments that fall within the scope of the Claims.

What is claimed is:

1. A cache with finely granular, programmable, locked-down regions comprising:

(a) a set associative array, each set having a plurality of lines, each line having at least one tag; and,

(b) the at least one tag having a field capable of indicating that at least a portion of the line is locked-down to prevent eviction and validity of each line or portion thereof.

2. The cache as recited in claim 1 wherein the field that indicates whether each line or portion thereof is locked-down is represented as a modified MESI state.

3. The cache as recited in claim 1 wherein the field that indicates whether each line or portion thereof is locked-down is programmed through at least one test register.

4. The cache as recited in claim 1 wherein each line is partitioned into at least two sectors.

5. The cache as recited in claim 1 wherein the finely granular, programmable, locked-down regions are set with high level commands through accesses to at least one model specific register.

6. The cache as recited in claim 1 wherein the field in the at least one tag indicates locked and invalid for each line or portion thereof that is defective.

7. A method for programmably locking-down finely granular regions in a cache comprising the steps of:

(a) caching data in a set associative array, each set having a plurality of lines, each line having at least one tag; and,

(b) programming a field in the at least one tag to indicate that at least a portion of the line is locked-down to prevent eviction and indicating validity of each line or portion thereof.

8. The method as recited in claim 7 wherein step (b) comprises writing a modified MESI state.

9. The method as recited in claim 7 wherein step (b) comprises programming the field through at least one test register.

10. The method as recited in claim 7 wherein step (b) comprises issuing high level commands to at least one model specific register.

11. The method as recited in claim 7 further comprising a step of indicating locked and invalid for each line or portion thereof that is defective.

12. In a computer having a motherboard, a processor having an internal cache, and system memory external to the processor, a method for programmably locking-down finely granular regions in the cache comprising the steps of:

(a) retrieving instructions from system memory;

(b) executing the instructions with the processor;

(c) caching instructions and data in the cache, the cache being a set associative array with each set having a plurality of lines and each line having at least one tag; and,

(d) responsive to predetermined instructions, programming a field in the at least one tag indicating that at least a portion of the line is locked-down to prevent eviction and indicating validity of each line or portion thereof.

13. The method as recited in claim 12 wherein step (d) comprises writing a modified MESI state.

14. The method as recited in claim 12 wherein step (d) comprises programming the field through at least one test register.

15. The method as recited in claim 12 wherein step (d) comprises issuing high level commands to at least one model specific register.

* * * * *

US005913224A

United States Patent [19]

MacDonald

[11] **Patent Number: 5,913,224**

[45] **Date of Patent: Jun. 15, 1999**

[54] **PROGRAMMABLE CACHE INCLUDING A NON-LOCKABLE DATA WAY AND A LOCKABLE DATA WAY CONFIGURED TO LOCK REAL-TIME DATA**

[75] Inventor: **James R. MacDonald**, Buda, Tex.

[73] Assignee: **Advanced Micro Devices, Inc.**, Sunnyvale, Calif.

[21] Appl. No.: **08/805,554**

[22] Filed: **Feb. 26, 1997**

[51] **Int. Cl.**[6] **G06F 12/14**

[52] **U.S. Cl.** **711/125**; 711/167; 711/145; 711/144

[58] **Field of Search** 711/144, 145, 711/167, 125

[56] **References Cited**

U.S. PATENT DOCUMENTS

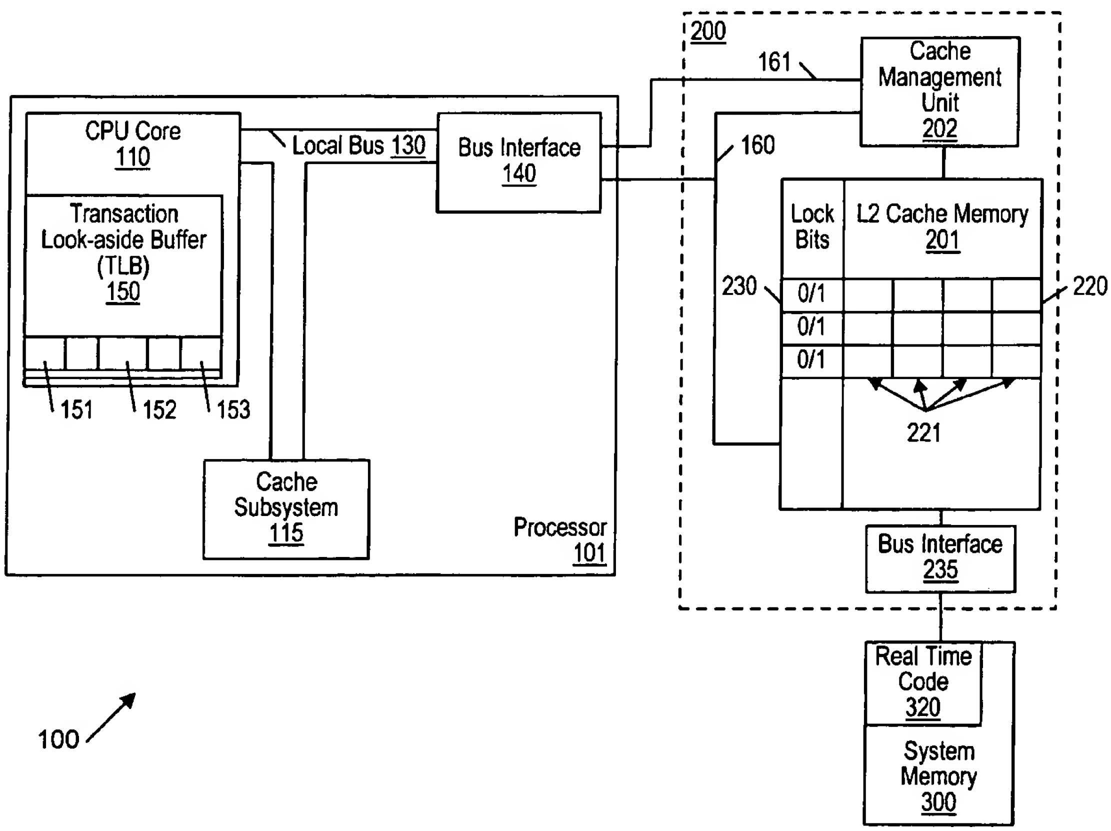
5,353,425 10/1994 Malamy et al. 711/144
5,487,162 1/1996 Tanaka et al. .
5,638,537 6/1997 Yamada et al. 711/167

Primary Examiner—Tod R. Swann
Assistant Examiner—David Langjahr
Attorney, Agent, or Firm—Conley, Rose & Tayon, PC; B. Noel Kivlin

[57] **ABSTRACT**

A computer system is disclosed which provides for execution of real-time code from cache memory. A cache management unit provides the real-time code to the cache memory from system memory upon a initiation of a read operation by a processor. Once in cache memory, the processor executes the real-time code from cache memory instead of system memory. The cache management unit detects read hits to cache each time the processor requests an instruction of code that is stored in the cache memory. Lock bits associated with each line of cache lock the contents of the line preventing the line from being overwritten under normal cache operation in which the least most recently used cached data is replaced by presently accessed data. Alternatively, one of a plurality of cache data ways may be dedicated to storing real-time code. Real-time code stored in the dedicated data way is not replaceable and thus is locked.

17 Claims, 5 Drawing Sheets



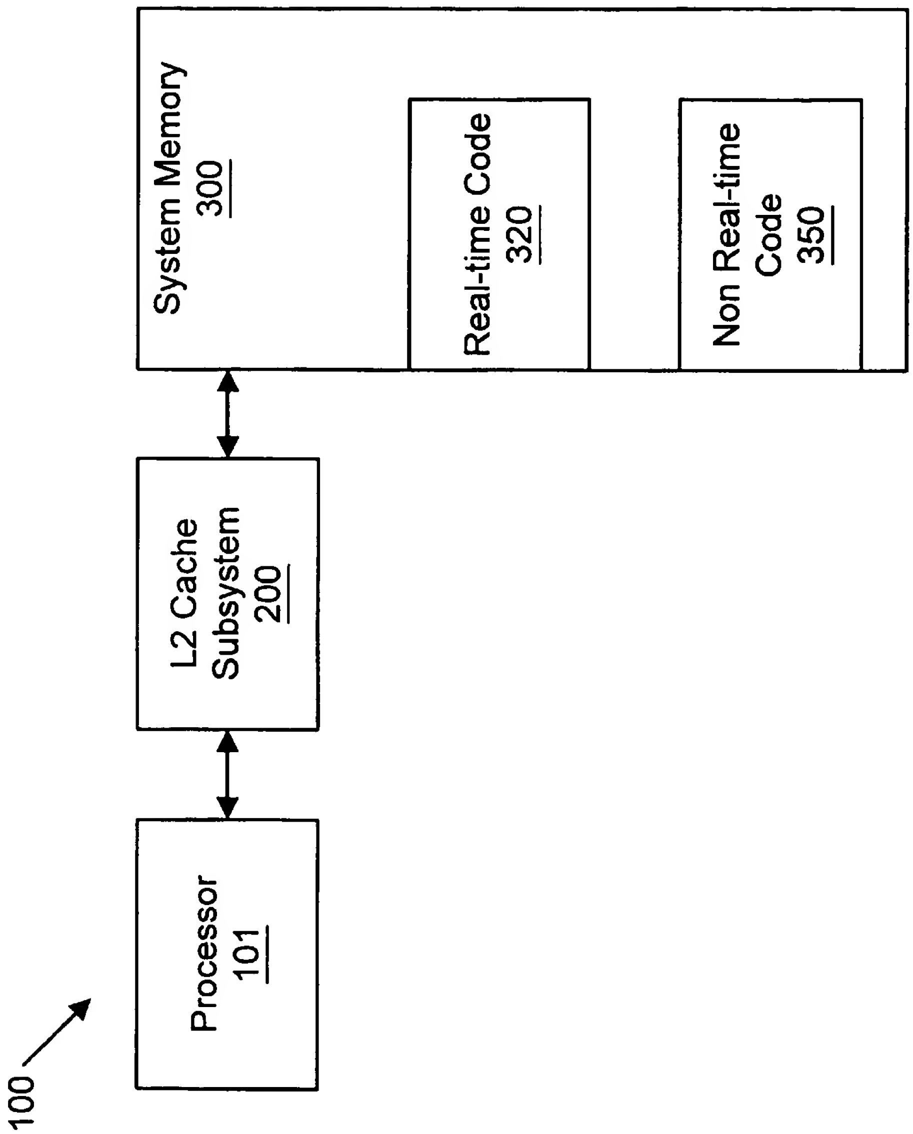
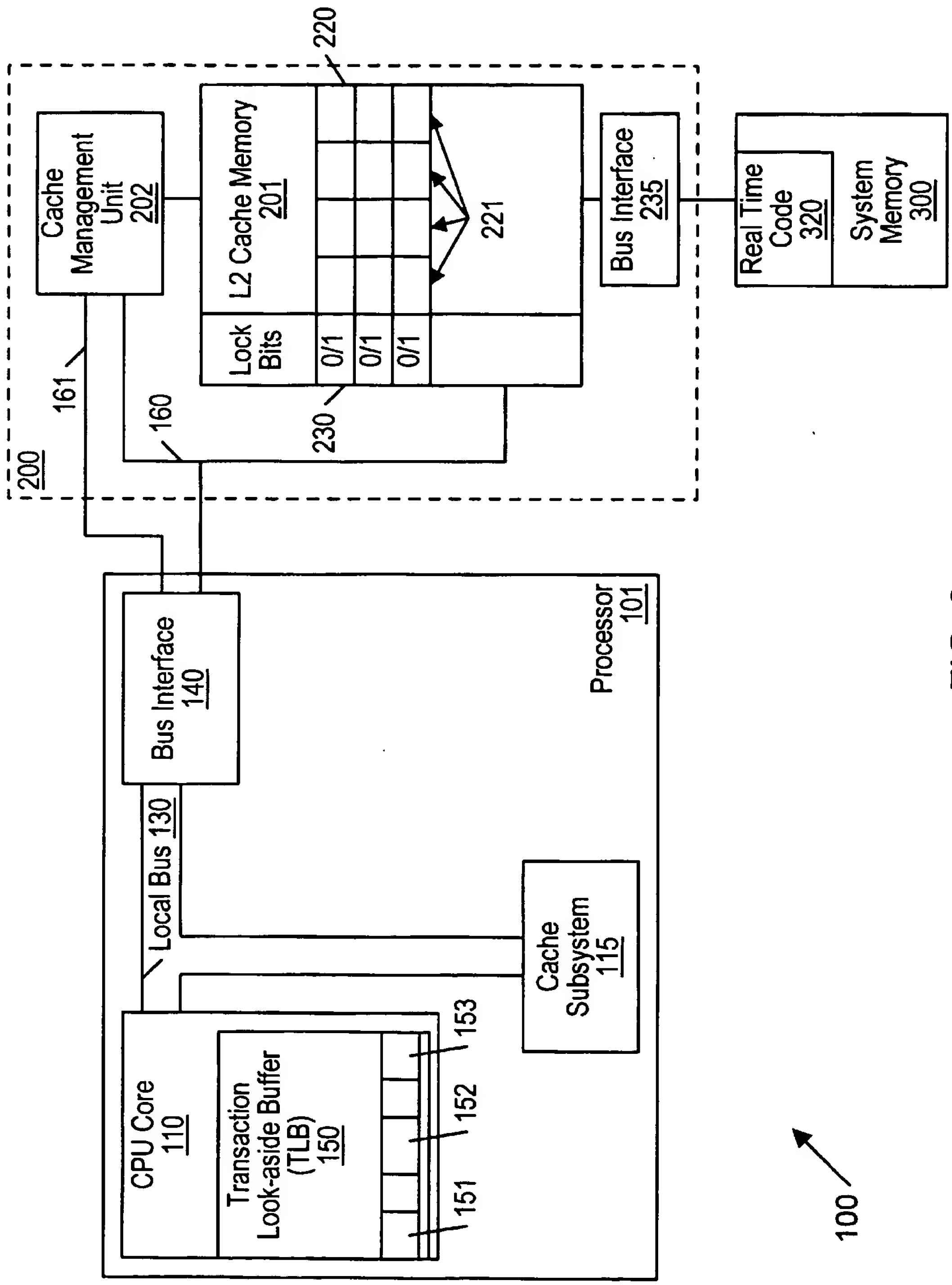

System Memory
300
Real-time Code
320
Non Real-time
Code
350
L2 Cache
Subsystem
200
Processor
101
100

FIG. 1



220
Cache Management Unit 202
L2 Cache Memory 201
Lock Bits
0/1
0/1
0/1
221
Bus Interface 235
Real Time Code 320
System Memory 300
161
160
230
200
Bus Interface 140
Processor 101
Local Bus 130
Cache Subsystem 115
CPU Core 110
Transaction Look-aside Buffer (TLB) 150
151
152
153
100

FIG. 2

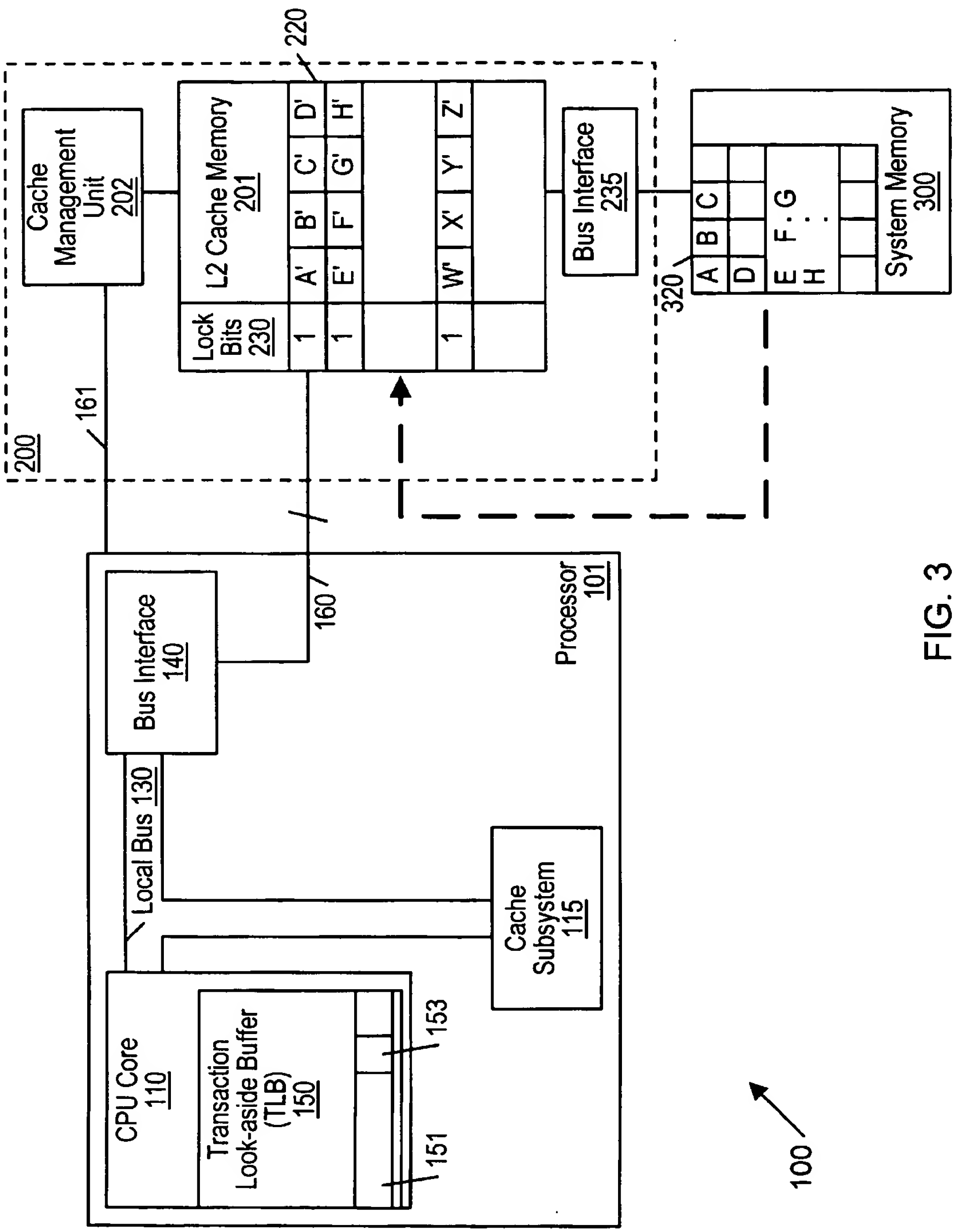

Cache Management Unit 202
L2 Cache Memory 201
220
Lock Bits 230
A' B' C' D'
E' F' G' H'
W' X' Y' Z'
1
1
1
Bus Interface 235
200
161
System Memory 300
A B C
D
E F G
H
320
Processor 101
Bus Interface 140
160
Local Bus 130
Cache Subsystem 115
CPU Core 110
Transaction Look-aside Buffer (TLB) 150
153
151
100

FIG. 3

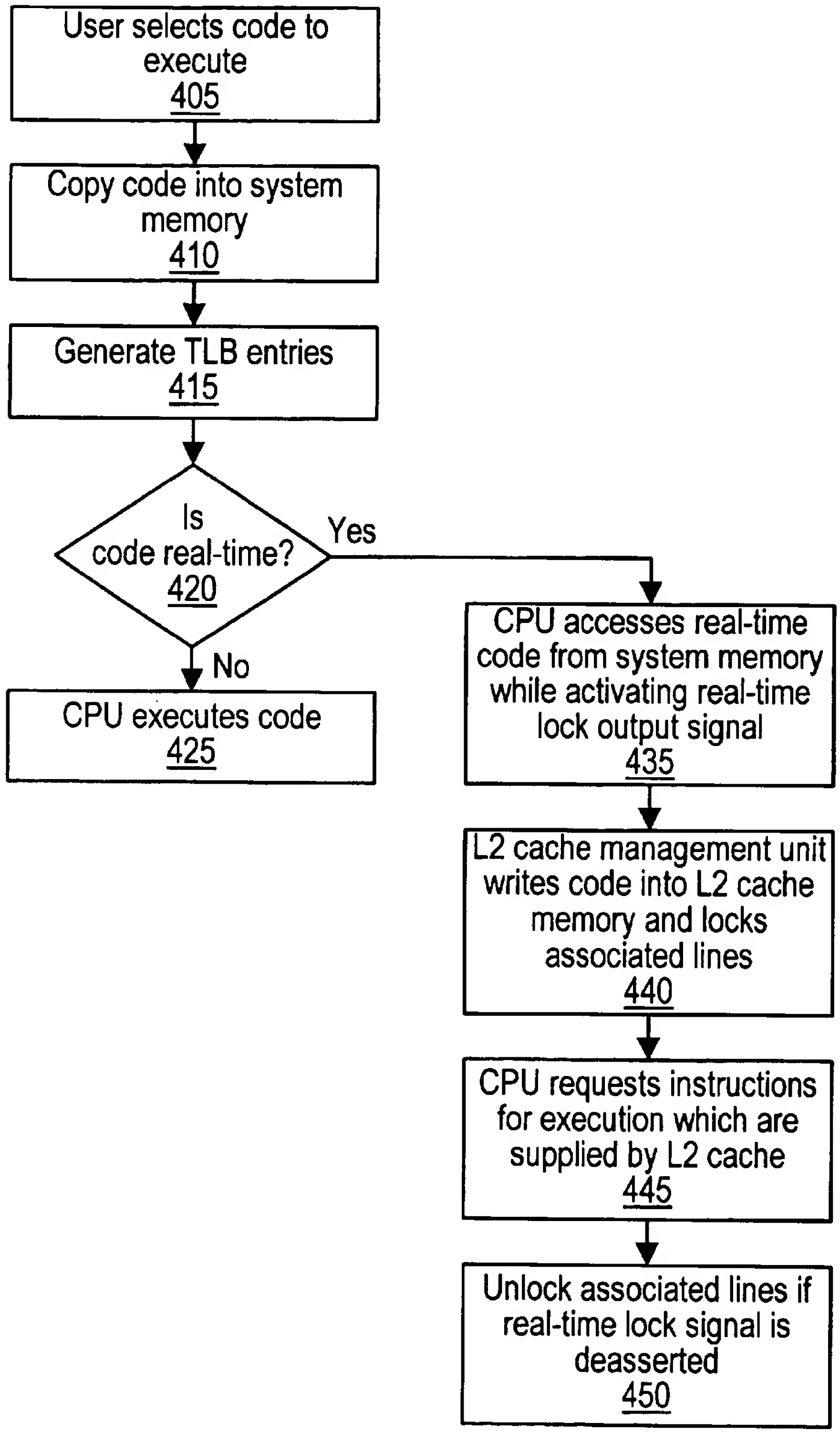

User selects code to execute 405
Copy code into system memory 410
Generate TLB entries 415
Is code real-time? 420
Yes
No
CPU executes code 425
CPU accesses real-time code from system memory while activating real-time lock output signal 435
L2 cache management unit writes code into L2 cache memory and locks associated lines 440
CPU requests instructions for execution which are supplied by L2 cache 445
Unlock associated lines if real-time lock signal is deasserted 450

FIG. 4

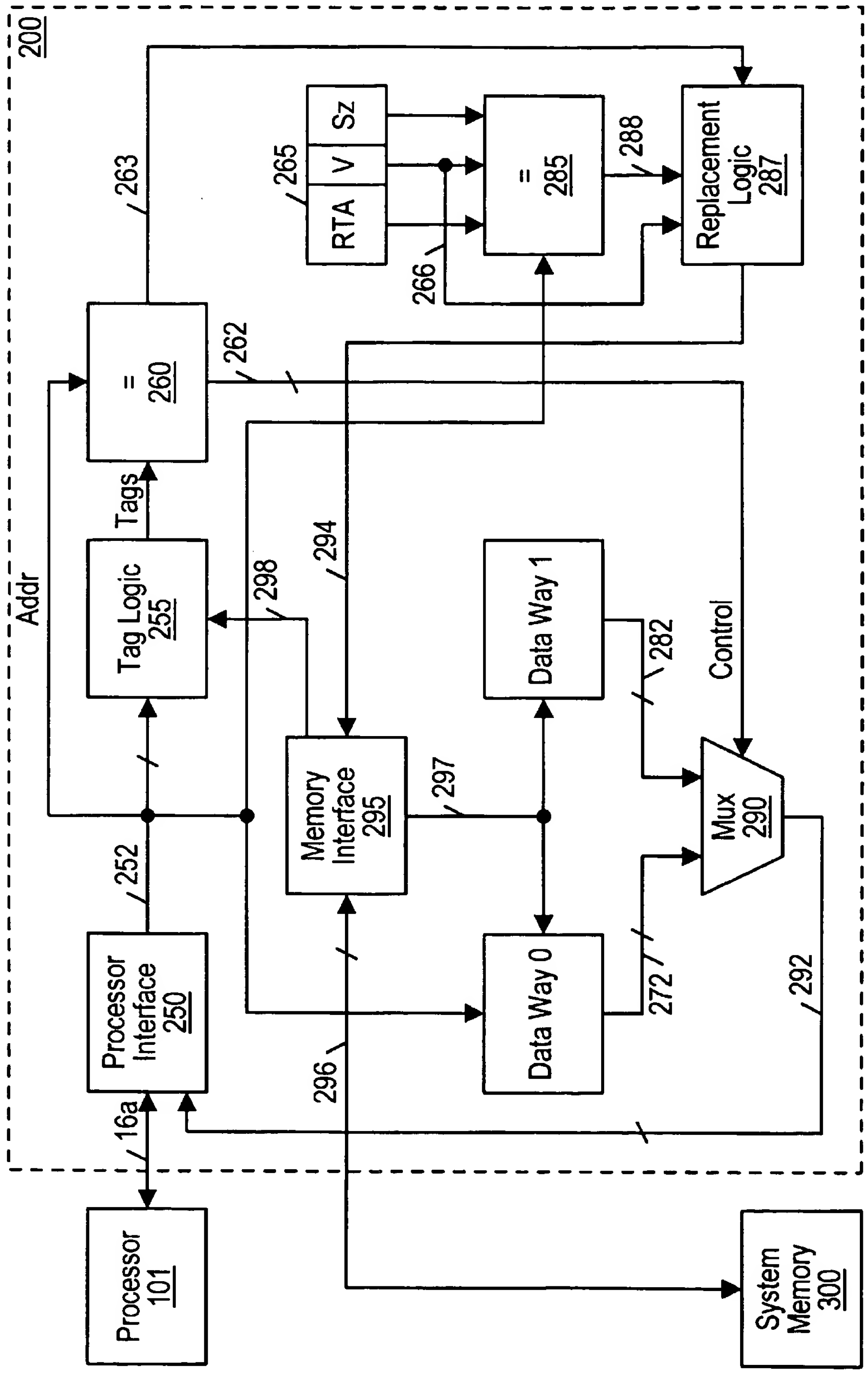

200
Addr
Tags
263
265
RTA
V
Sz
=
285
288
Replacement Logic
287
266
262
=
260
Tag Logic
255
298
294
Data Way 1
282
Control
Memory Interface
295
297
Mux
290
252
Processor Interface
250
Data Way 0
272
292
296
16a
Processor
101
System Memory
300

FIG. 5

PROGRAMMABLE CACHE INCLUDING A NON-LOCKABLE DATA WAY AND A LOCKABLE DATA WAY CONFIGURED TO LOCK REAL-TIME DATA

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to more efficient execution of software in computer systems. More particularly, the present invention relates to computer systems for executing software stored in cache memory subsystems. Still more particularly, the present invention relates to cache subsystems that load real-time event processing software for more efficient execution.

2. Description of the Relevant Art

Software to be executed by a microprocessor typically is stored on a floppy or fixed disk medium. Once a request is made by a user to execute a program, the program is loaded into the computer's system memory which usually comprises dynamic random access memory devices (DRAM). The processor then executes the code by fetching an instruction from system memory, receiving the instruction over a system bus, performing the function dictated by the instruction, fetching the next instruction, and so on.

Generally, whenever system memory is accessed, there is a potential for delay between the time the request to memory is made (either to read or write data) and the time when the memory access is completed. This delay is referred to as "latency" and can limit the performance of the computer.

There are many sources of latency. For example, operational constraints with respect to DRAM devices cause latency. Specifically, the speed of memory circuits is based upon two timing parameters. The first parameter is memory access time, which is the minimum time required by the memory circuit to set up a memory address and produce or capture data on or from the data bus. The second parameter is memory cycle time, which is the minimum time required between two consecutive accesses to a memory circuit. For DRAM circuits, the cycle time typically is approximately twice the access time. DRAM circuits today generally have access times in the approximate range of 60–100 nanoseconds, with cycle times of 120–200 nanoseconds. The extra time required for consecutive memory accesses in a DRAM circuit is necessary because the internal memory circuits require additional time to recharge (or "precharge") to accurately produce data signals. Thus, even a processor running as slow as 10 MHz cannot execute two memory accesses in immediate succession (i.e., with adjacent clock pulses) to the same 100 nanosecond DRAM chip, despite the fact that a clock pulse in such a microprocessor is generated every 100 nanoseconds. A DRAM chip requires time to stabilize before the next address in that chip can be accessed. Consequently, in such a situation the processor must wait by executing one or more loop cycles before it can again access data in the DRAM circuit. Typically, a memory controller unit ("MCU") is provided as part of the computer system to regulate accesses to the DRAM main memory. Latency caused by long memory cycle times relative to processor speeds has become a particularly acute problem today as processor speeds in excess of 100 MHz are commonplace. Instead of waiting one or two clock cycles to again access a 100 nanosecond DRAM device, today's "486" and "Pentium" processors must wait 20 or more clock cycles.

In addition to the delays caused by access and cycle times, DRAM circuits also require periodic refresh cycles to protect the integrity of the stored data. These cycles consume approximately 5 to 10% of the time available for memory accesses, and typically are required approximately every 4 milliseconds. If the DRAM circuit is not refreshed periodically, the data stored in the DRAM circuit will be lost. Thus, memory accesses may be halted while a refresh cycle is performed.

Further, most, if not all, computer architectures today include multiple bus master systems. Any one of a number of bus masters may obtain ownership or control of the system bus and thereby access system memory. Normally, granting a bus master device ownership of the system bus, from among competing requests for ownership, is based on a predetermined hierarchy. In a hierarchy scheme, one bus master device may have a higher position in the hierarchy than another bus master device. Accordingly, the former device would be granted ownership of the system bus if there was a conflict between the two devices where each device contemporaneously sought control of the bus. Although hierarchy schemes are valuable for resolving conflicts between multiple bus master devices requesting control of the bus to access system memory, such schemes force a bus master that must yield to a higher priority bus master to wait while the other device executes its memory transaction, thereby causing latency with respect to the waiting device.

The latency associated with memory accesses may be different and unpredictable from one memory access to the next. For many software applications unpredictable latency is not a significant problem. However, for core sequences, especially those related to real-time event processing such as music synthesis which implement digital signal processing, unpredictable latency can greatly interfere with proper performance and produce undesirable results.

To expedite memory transfers, most computer systems today incorporate cache memory subsystems. Cache memory is a high-speed memory unit interposed between a slower system DRAM memory and a processor. Cache memory devices usually have speeds comparable to the speed of the processor and are much faster than system DRAM memory. The cache concept anticipates the likely reuse by the microprocessor of selected data in system memory by storing a copy of the selected data in the cache memory. When a read request is initiated by the processor for data, a cache controller determines whether the requested information resides in the cache memory. If the information is not in the cache, then the system memory is accessed for the data and a copy of the data may be written to the cache for possible subsequent use. If, however, the information resides in the cache, it is retrieved from the cache and given to the processor. Retrieving data from cache advantageously is faster than retrieving data from system memory, involving both less latency and more predictable latency.

Code, as well as data, is subject to being stored in cache. Cache memory size, however, is generally much smaller than system memory and is used only to store the most recently used data or code anticipating the reuse of that information. Because the cache is relatively small and is used only for storing the most recently accessed code or data, old code or data (i.e., less recently used code or data in cache) is at risk of being overwritten by new code or data. Although replacement generally causes no problem for many types of data and code, replacement of real-time code can detrimentally affect the predictability of the latency of accesses to the real-time code or data and thus may cause improper or poor multimedia performance.

BRIEF SUMMARY OF THE INVENTION

The problems outlined above are in large part solved by the teachings of the present invention. The present invention

relates to a system and method for locking real-time code into a cache memory to avoid repetitively accessing the real-time code from system memory. A processor reads the real-time code and the cache subsystem writes the code into an entry of the cache upon detecting a read miss. An output signal from the processor during the reading of the real-time code indicates to the cache subsystem the real-time nature of the code. In response, the cache subsystem locks the code into the cache preventing overwriting the code with more recently used data. The real-time code is locked into cache by setting a lock bit associated with each line of cache containing the real-time code. Once stored and locked into the cache subsystem, the processor fetches instructions for execution. Because the instructions have been stored in cache, the cache subsystem, according to normal cache protocol, supplies the requested instructions.

After the processor has completed execution of the real-time code from the cache subsystem, the processor may direct the cache subsystem to unlock the previously executed real-time code to allow for other real-time code modules to be executed from cache memory. Unlocking real-time code is accomplished by clearing the lock bits associated with the lines containing real-time code.

An alternative to setting lock bits associated with each line of cache containing real-time code includes a real-time address register that generally defines which system memory addresses contain real-time code. The register preferably includes a starting address of the real-time code and a size value representing the number of addresses containing the real-time code. The register also includes a valid bit to indicate whether the real-time locking feature of the invention is turned on or off. When the valid bit is off, all information, including real-time code, is stored in cache according normal cache behavior. However, when the valid bit is on, nonreal-time code is stored in a first way in the cache and real-time code is stored in second cache way. Real-time code stored in the second cache way is not replaced and thus is locked into the cache. To unlock real-time code in the alternative embodiment, the valid bits in the real-time register is cleared.

BRIEF DESCRIPTION OF THE DRAWINGS

Other objects and advantages of the invention will become apparent upon reading the following detailed description and upon reference to the accompanying drawings in which:

FIG. 1 is a block diagram representation of a typical computer system;

FIG. 2 is a block diagram of the computer system consistent with the preferred embodiment for locking real-time code into cache;

FIG. 3 is a block diagram of the preferred computer system showing the data flow of real-time code into cache;

FIG. 4 is a flow chart outlining the steps in executing real-time code from cache memory; and

FIG. 5 is a block diagram showing an alternative embodiment for locking real-time code in cache.

While the invention is susceptible to various modifications and alternative forms, specific embodiments thereof are shown by way of example in the drawings and will herein be described in detail. It should be understood, however, that the drawing and detailed description thereto are not intended to limit the invention to the particular form disclosed, but on the contrary, the intention is to cover all modifications, equivalents and alternatives falling within the spirit and scope of the present invention as defined by the appended claims.

DETAILED DESCRIPTION OF THE INVENTION

Turning now to the drawings, FIG. **1** is a block diagram of a computer system **100** to which the present invention is adapted. A processor **101** couples to a system memory unit **300** via an L2 cache subsystem **200**. Information stored in system memory **300** is accessible by the processor **101** and accessed data preferably is made available to L2 cache subsystem **200** for temporary storage consistent with commonly known cache techniques. As one of ordinary skill in the art will recognize, many different types of information is stored in system memory **300** including real-time code **320** and non-real-time code **350**. Processor **101** may fetch code from an internal cache, L2 cache subsystem **200** or system memory **300**. As described in greater detail below, however, processor **101** preferably executes real-time code **320** from the L2 cache subsystem, rather than system memory **300**, thereby reducing deleterious latency effects on real-time computations.

Referring now to FIG. **2**, the processor **101**, L2 cache subsystem **200**, and system memory **300** is shown in greater detail. Consistent with the preferred embodiment, the processor **101** includes a CPU core **110** coupled to an internal cache memory subsystem **115** and a local bus interface **140** via local bus **130**. Local bus **130** has a predetermined bit width and is the processor's primary bus. Bus interface **140** provides processor **101** with an interface to L2 cache subsystem **200** over lines **160**, **161**. The CPU core **110** also includes a translation lookaside table ("TLB") **150**.

As illustrated, computer system **100** embodies a single processor. It is understood, however, that the present invention may be adapted to multi-processor systems. CPU core **110** is a data processing unit that implements a predetermined instruction set. Exemplary processing units include models 80386, 80486, and Pentium microprocessors. The present invention should not be limited to any particular processing units.

The TLB **150** generally comprises a cacheable set of table entries **151** to provide translations between virtual addresses and physical addresses, as one of ordinary skill in the art would know. Normally, a page address is derived from the upper order bits of the virtual address and used to access physical page addresses in the TLB. Pages range in size, but 4K bytes is typical. Also, stored in each table entry are various attributes such as read/write bits **152** for indicating whether the data stored at the associated physical address is read-only, write-only, or both. Consistent with the preferred embodiment, the TLB **150** includes with each entry **151** a real-time code bit **153** that specifies whether the information stored at the associated physical page address includes real-time code or not. The real-time code bit **153** is written when the relevant table entry is created by the operating system during commonly known allocation schemes. It is noted that the table entries are only temporarily stored in TLB **150**. A table in memory preferably is used to store table entries.

The TLB real-time lock bit **153** facilitates real-time code to be stored in L2 cache without being overwritten through normal cache replacement behavior. Because cache subsystems typically overwrite their contents based on a least recently used scheme, real-time code executed from cache is at risk for being overwritten before it is completely executed. To avoid overwriting real-time code, the L2 cache is able to lock in specified contents.

The L2 cache subsystem 200 preferably includes an L2 cache memory 201 coupled to a cache management unit 202 for directing the transfers of data into and out of the L2 cache memory 201. Cache management unit 202 also controls and orchestrates the transfer of data, address and control signals between local bus 130 and system memory 300. Cache management unit 202 preferably includes a memory controller for providing access to L2 cache memory 201. The memory controller may be any one of a number of commonly known memory controllers compatible with the selected CPU core 110 and overall computer architecture. Such a memory controller may be located as part of the processor 101. Processor 101 also includes a real-time lock output signal 161 preferably provided to the cache management unit 202. The real-time lock signal 161 indicates that the information in system memory 300 requested by processor 101 includes real-time code. This feature will be explained in more depth below. L2 cache subsystem 200 also includes a bus interface 235 which provides an interface to system memory 300.

L2 cache memory 201 includes a plurality of cache lines 220. Associated with each line of L2 cache memory 201 is address tag and state information (not specifically shown). The address tag indicates a physical address in system memory 300 corresponding to each entry within cache memory 201. In this embodiment each entry within L2 cache memory 201 is capable of storing a line of data. A line of data preferably consists of four double words 221 (where each double word comprises 32 bits). It is understood, however, that a line could contain any number of word or double words, depending upon the system. It is further understood that a double word could consist of any number of bits.

The state information is comprised of a valid bit and a set of dirty bits. A separate dirty bit is allocated for each double word within each line. A valid bit indicates whether a predetermined cache line contains valid cache data, while the dirty bits identify the write status of each double word within each cache line. In an invalid state, there is no valid data in the corresponding cache memory entry. In a valid and clean state, the cache memory entry contains data which is consistent with system memory 300. In a valid and dirty state, the cache memory entry contains valid data which is inconsistent with system memory 300. Typically, the dirty state results when a cache memory entry is altered by a write operation.

Cache management unit 202 includes an address tag and state logic circuit (not specifically shown) that contains and manages the address tag and state information. A comparator circuit for determining whether a cache hit has occurred, and a snoop write-back circuit that controls the write back of dirty data within L2 cache memory 201. It will be appreciated by those skilled in the art that cache management unit 202 may contain additional conventional circuits to control well-known caching functions such as various read, write, update, invalidate, copy-back, and flush operations. Such circuitry may be implemented using a variety of configurations.

In one embodiment, L2 cache subsystem 200 comprises a set associative cache configuration. Least recently used replacement may be employed to select one of the ways for replacement.

System memory 300 is a physical memory device of a predetermined size and may be implemented with DRAM (dynamic random access memory). System memory 300 may be used to store data, code, and the like. The code stored in system memory 300 includes real-time 320. Multiple real-time code modules may be stored in system memory 300.

Referring still to FIG. 2, cache memory 201 includes a plurality of lines of data 220. Preferably associated with each line of data is a lock bit 230. The lock bit can be set to lock the associated line of data. Once locked, the line of data cannot be overwritten pursuant to normal cache behavior in which the least recently used cache line is overwritten by new data to be stored in the cache. The lock bit overrides the least recently used replacement scheme for the line associated with the lock bit. A "0" value for the lock bit indicates that the associated line of data is not locked, where as a lock bit value of "1" indicates that the line of data is locked. The logic level of the lock bits, of course, can be reversed, i.e. a "0" value indicating the associated line is locked and a "1" value indicating that the associated line is not locked. For purposes of the following discussion, it is assumed that a logic "1" lock bit value indicates the locked condition. To specify which cache contents to lock, computer system 100 asserts its real-time lock output signal on line 161 to indicate to the cache management unit 202 when to lock data in cache.

Consistent with the preferred embodiment, generally four major steps facilitate the execution of real-time code from L2 cache. These steps presuppose that the targeted real-time code module has already been written by the operating system into the system memory 300 from a disk or other medium on which the code was stored. First, the processor directs the entire block of real-time code to be stored in the L2 cache memory 201 while indicating to the cache management unit 202 that the information being stored in cache comprises real-time code, as opposed to non-real-time code, data, or other types of information. Second, the L2 cache subsystem 200 locks the real-time code into the L2 cache memory 201 to avoid overwriting. Third, the processor 101 executes the real-time code after it has been stored in L2 cache memory 201. Lastly, after the processor has completed its execution of the real-time code from L2 cache memory and no longer needs access to the code, the L2 cache subsystem unlocks the real-time code freeing up that part of L2 cache memory for other real-time code modules.

Software to be executed by CPU core 110 normally is transferred from a disk to system memory 300 and then fetched from system memory 300 by CPU core 110 through the L2 cache subsystem 200 and lines 160. Consistent with the preferred embodiment, computer system 100 takes advantage of the L2 cache memory's lock bits, the TLB real-time code bit, and the processor's real-time lock output signal 161 to allow execution of real-time code 320 from L2 cache memory 201, instead of system memory 300.

Referring now to FIG. 3, a block diagram illustrating the flow of data within computer system 100 to transfer real-time code 320 from system memory 300 to cache memory 201 is shown. Real-time code 320 consists of a plurality of double words, as exemplified by double words "A" through "Z." Before the CPU core 110 executes real-time code 320 from L2 cache memory 201, the CPU core 110 must have the real-time code 320 transferred from system memory 300 to L2 cache memory 201. This process preferably is accomplished by a read operation by the CPU code 110 of all of the double words in system memory 300 comprising real-time code 320. Because the real-time code 320 does not already exist in the L2 cache memory 201 when the CPU 110 reads the real-time code 320 for the first time, the cache management unit 202 detects a read miss and directs the real-time code to be transferred into L2 cache memory 201 pursuant

to normal L2 cache behavior. A copy of the real-time code thus is placed in L2 cache memory **201** as indicated by the lines of L2 cache memory comprising double words "A" through "Z".

These lines of cache containing real-time code ultimately must be locked to prevent replacement. The L2 cache subsystem **200**, therefore, must be made aware which of its contents include real-time code and which do not. The processor **101** provides this indication by asserting the real-time lock output signal while reading the code from system memory **300**. This signal indicates to the cache management unit **202** that it must lock the lines of cache in which it writes the associated real-time code.

The following discussion describes how the processor **101** determines that the code it requests for executing is real-time code. As stated, the TLB **150** includes a real-time code bit for each entry. Thus, when the processor initiates a read request and translates the requested virtual address to a physical address by accessing the TLB, the CPU core **110** reads the associated real-time code bit **153**. If the bit is set to indicate that the requested information is real-time code, the processor asserts its real-time lock output signal on line **161** while also asserting the address and data signals to effectuate a read cycle.

Upon detecting a read miss while the processor's real-time lock output signal is asserted, the cache management unit **202** writes the requested real-time code **320** to L2 cache memory **201** and sets the lock bit associated with that line to a logic "1" indicating that this line of cache memory cannot be overwritten by subsequent cache replacement activity. Alternatively, the cache management **202** may wait until the entire real-time code **320** is stored in L2 cache memory **201** before setting all of the lock bits to a logic "1" level. Once the real-time code is completely stored in L2 cache memory **201** and all of the associated lock bits are set, the CPU core **110** then can execute the real-time code **320**. At this point, as one of ordinary skill in the art will readily understand, it is transparent to the CPU core **110** that execution of the real-time code is from L2 cache memory **201** instead of from system memory **300**. The CPU core **110** fetches each instruction of the real-time event handler by issuing physical addresses pertaining to locations in system memory **300** of the real-time code **320**. The cache management unit **202**, however, detects a read hit as the requested instruction of the real-time code is also stored in L2 cache memory **201**. In response, the cache management unit **202** directs the requested instruction to be supplied to the CPU core **110** from L2 cache memory **201** instead of from system memory **300**. In this manner, the real-time event handler is executed by the CPU core **110** from L2 cache memory **201** with reduced latency and increased latency predictability.

Once the real-time code **320** is completely executed, it may be desired to unlock the real-time code **320** from L2 cache memory, thus freeing up cache entries for other code or data. Cache lines are unlocked by changing the state of the lock bit associated with the targeted lines. In computer systems consistent with the preferred embodiment, invalidate or flush operations preferably are used to unlock cache entries. Invalidating the cache preferably is initiated upon operating system reallocation of the page corresponding to the real-time code, as one of ordinary skill in the art would understand. For example, when a page is selected for reallocation, if the current translation to the page has the real-time code bit set, the operating system may execute a flush operation to each line in the page. L2 subsystem **200** resets the lock bit for the corresponding line. Alternatively, a flush operation indicated to be a real-time operation via real-time signal **161** may cause all of the lock bits to be reset.

FIG. **4** shows a flow diagram exemplifying a method consistent with the preferred embodiment for executing real-time code from L2 cache memory. In step **405**, the real-time code module to be executed is selected and copied into the system memory by the operating system (step **410**). Page allocations and TLB entries in step **415** are updated and if the code copied into system memory is a real-time module, the real-time bit in the corresponding TLB entry is updated. The CPU determines whether the code is real-time or not in step **420** by accessing and checking the state of the real-time code bit corresponding to the real-time code. If the code is real-time, the CPU accesses the entire code in step **435**. Upon detecting a read miss in step **440**, the L2 cache subsystem writes the real-time code into cache and locks the associated lines. In step **445**, the CPU may then execute the real-time code by fetching instructions which are provided by the L2 cache where the code is stored. Finally, in step **450**, the lines of cache memory that were used to store the real-time code are unlocked as described above. If however, the code is not real-time (step **420**), the CPU fetches the code from memory and executes it according to known protocols.

It is noted that while the L2 cache in FIGS. **1–3** is interposed between the processor **101** and system memory **300**, other cache configurations are possible. For example, the cache may comprise a look aside configuration or backside cache configuration in which a system memory bus couples the CPU, cache, and system memory.

Referring now to FIG. **5**, an alternative embodiment for locking real-time code into cache is shown to comprise L2 cache subsystem **200** coupled to processor **101** over lines **160** and to system memory **300**. L2 subsystem **200** includes a processor interface **250**, tag logic **255**, memory interface **295**, data way **0**, data way **1**, multiplexer **290**, comparators **260** and **285**, real-time address registers **265**, and replacement logic **287**. Processor interface **250** couples to tag logic **255**, data way **0**, and comparators **260**, **285** over lines **252**. Tag logic**255** provides tag information to comparator **260** which compares the address signals provided by processor interface **250** to the tag information provided by tag logic **255** to determine the existence of a cache hit or miss, as one of ordinary skill in the art would know. Comparator **260** provides an output signal on line **262** to multiplexer **290** and another output signal on line **263** to replacement logic **287**. The output signal provided to multiplexer **290** is asserted upon detecting a cache hit by comparator **260** to select the requested data from the data way that contains the requested data. The output signal on line **263** indicates the presence of a cache miss to replacement logic **287**.

Memory interface **295** directs the operation of data way **0** and data way **1** via lines **297** dining cache hits and misses and also provides communication with system memory **300** over lines **296** for retrieving data from system memory **300** to be stored in one of the two data ways. Although only two data ways are shown in FIG. **5**, one of ordinary skill in the art will recognize that the invention could include additional data ways. The output signals from data way **0** and data way **1** over lines **272**, **282**, respectively are provided to multiplexer **290**. Multiplexer **290** is a known 2:1 multiplexer in which one of two input signals is provided as an output signal in response to the state of a control signal. During a cache miss, multiplexer **290** is controlled by the output signal from comparator **260** on line **262**. The signal on line **262** determines which of the two input signals on lines **272** and **282** are to be selected by multiplexer **290** as an output signal on line **292**. The output signal of multiplexer **290** is provided to processor interface **250**.

During a cache miss, memory interface **295** is controlled by replacement logic **287** to store data corresponding to the

new address (i.e., the address for which there was a cache miss) in one of the data ways. Memory interface **295** retrieves the requested data from system memory over lines **296** and stores the data in one of the two data ways as determined by replacement logic **287** in accordance with the present invention.

Real time address register **265**, although shown as a single register in FIG. **5**, may include multiple registers. Each register preferably includes a real-time address field (RTA), a valid bit field (V), and a size field (Sz) associated with a real-time code module. The real-time address field preferably includes the starting address of the real-time code. The size field indicates the size of the real time code whose starting address is specified in the RTA field of register **265**. The RTA and Sz fields thus specificy the location of real-time code in system memory. As explained below, by comparing and address from processor **101** to the contents of register **265**, comparator **285** can determine whether the address is an address pertaining to real-time code. The V bit indicates whether the real time code locking feature of the present invention associated with the real time code beginning at the RTA address is enabled or disabled. Thus, a V bit that is set indicates that the real-time code locking feature is turned on (enabled) for the real-time code specified by the RTA and Sz fields. Conversely, the real-time code locking feature can be turned off (disabled) by clearing the V bit. The contents of the real-time address register preferably is initiated by the computer's operating system or a device driver as one of ordinary skill will recognize.

It should be recognized that the contents of register **265** generally define which system memory addresses contain real-time code. Thus, register **265** could be configured differently than that described above. For example, register **265** could include a beginning and an ending address instead of a beginning address and a size value.

The contents of real-time address register **265** is provided to comparator **285** which compares the contents of the real-time address register **265** to the address provided by processor interface **250**. Comparator **285** provides an output signal to replacement logic **287** over lines **288** to indicate whether the address from processor interface **250** is an address corresponding to real-time code, or not. Replacement logic **287** provides control signals to memory interface **295** over lines **294** generally for directing the storage into cache of real-time code in accordance with the present invention.

The operation of the alternative embodiment shown in FIG. **5** will now be described with reference to four situations—(1) valid bit not set, cache miss, (2) valid bit set, cache miss, and address within the real-time address range specified by register **265**, (3) valid bit set, cache miss, and address not within real-time address range specified by register **265**, and (4) cache hit. In the second and third situations, it will be seen that data way **1** is used to store real-time code and data way **0** is used to store nonreal-time code. However, the selection of which data way to use for storing real-time code is not important. Thus, data way **0** could be used to store real-time code.

In the first situation in which a cache miss is detected and the V bits of registers **265** are cleared indicating that the real-time code locking feature of the present invention has been turned off, L2 cache subsystem **200** functions in accordance with known cache protocol. An address provided to processor interface **250** is compared against the tags stored in tag logic **255**. Upon detection of cache miss by comparator **260** (i.e., the data corresponding to the address provided by processor interface **250** is not currently stored in either data way), a signal on line **263** directs the replacement logic **287** to store data corresponding to that address in one of the data ways. The V bits from register **265** are also provided to replacement logic **287** on line **266** and thus replacement logic **287** can determine that the locking feature of the invention is disabled. In this situation (cache miss, V bits clear), the replacement logic **287** stores the data corresponding to the address from processor interface **250** in either data way in accordance with known protocols. The data to be stored in cache is retrieved from system memory **300** over lines **296** by memory interface **295** and stored in the selected data way. Replacement logic **287** may use in any commonly known replacement algorithm such as the least recently used algorithm in which the least recently used datum in the data ways is replaced by the new data. Tag logic **255** is then updated by memory interface **295** to include the tag associated with the new data stored in the data ways.

In the second situation, at least one of the valid bits in registers **265** is set indicating that the real-time locking feature of the present invention is enabled, and a cache miss occurs. A cache miss for a real-time code address results in the storage of the associated real-time code in data way **1**. Comparator **260** compares the address to tags from tag logic **255** and indicates the existence of a cache miss on line **263** to replacement logic **287**. Comparator **285** consequently compares the address from processor interface **250** to the range of real-time addressed specified by registers **265** and determines that the address from processor interface **250** falls within the range of real-time addresses. Comparator **285** provides a signal on line **288** to replacement logic **287** indicating that the new address pertains to real-time code. In response, replacement logic **287** directs the memory interface **250** to retrieve the real-time code associated with the current address. After retrieval of the real time code, the real-time code is stored in data way **1** without replacing any other real-time code already stored in data way **1**. As explained previously, data way **1** is dedicated to the storage of real-time code when a V bit is set. Replacement logic **287** and memory interface **295** cooperate to prevent any real-time code from being replaced when the real-time code locking feature is enabled.

The third situation is similar to the second situation except that the address received by processor interface **250** is not an address for real-time code. Comparator **260** detects a cache miss and comparator **285** determines that the new address does not lie within the real-time address range specified by registers **265**, and that at least one V bit is set indicating that the real-time code feature of the present invention is enabled. Comparator **285** indicates to replacement logic **287** on lines **288** that the address is not a real-time code address. In response, replacement logic **287** directs the memory interface to retrieve the data corresponding to the address from system memory **300** and store it in data way **0** preferably according to the least recently used algorithm described previously.

In the fourth situation comparator **260** detects a cache hit upon comparing the address from processor interface **250** and tags from tag logic **255**. The output signal from comparator **260** on line **262** is asserted indicating the presence of a cache hit and also indicates in which of the data ways the requested data is located. Multiplexer **290** uses this output signal as a control signal and provides on its output lines **292** the data from the data way specified by the state of the control signal. The requested data is provided to processor **101** through processor interface **250**.

It should be recognized that the real-time code locking feature of the present invention can be disabled by simply

clearing all of the V bits in registers **265**. Once the V bits are cleared, cache storage proceeds in accordance with known protocols and new data can be stored in either data way according to, for example, the least recently used method.

Numerous variations and modifications will become apparent to those skilled in the art once the above disclosure is fully appreciated. For example, cache subsystems internal to the processor (so called "L1 caches") may be used for storing, locking, and executing real-time code. It is intended that the following claims be interpreted to embrace all such variations and modifications.

What is claimed is:

1. A computer system for executing code from cache memory comprising:

a processor for executing code;

a system memory device for storing code and data;

a system bus coupling said processor and said system memory;

a cache memory subsystem coupled to said system bus, wherein said cache memory subsystem includes a plurality of data ways, wherein each data way includes a plurality of cache entries each configured for temporary storage of a line of code or data, wherein the cache entries of at least one of said plurality of data ways are lockable to lock real-time code therein, and wherein the cache entries of at least another of said plurality of data ways are not lockable.

2. The computer system of claim **1**, wherein said processor includes a translation lookaside buffer (TLB) which includes a real-time code bit which indicates whether information stored in said system memory device corresponding to the real-time code bit comprises real-time code.

3. The computer system of claim **2**, wherein said processor is configured to assert a real-time code output signal.

4. The computer system of claim **3**, wherein real-time code is stored and locked in said at least one of said plurality of data ways of said cache memory subsystem upon assertion of said real-time code output signal by said processor during a read operation by said processor.

5. The computer system of claim **4**, wherein said cache memory subsystem upon storing said real-time code in said at least one of said plurality of data ways of said cache memory subsystem, locks said lines of cache in which said real-time code is stored in response to receiving said real-time code output signal from said processor.

6. A method of executing real-time code from cache memory, wherein said cache memory includes a plurality of data ways, wherein each data way includes a plurality of cache entries each configured for temporary storage of a line of code or data, wherein the cache entries of at least one of said plurality of data ways are lockable to lock real-time code therein, and wherein the cache entries of at least another of said plurality of data ways are not lockable, the method comprising the steps of:

(a) updating entries in a TLB for translating virtual addresses associated with real-time code to physical addresses;

(b) further updating a real-time code bit to ascertain if code to be executed comprises real-time code;

(c) reading said real-time code bit to ascertain if code to be executed comprises real-time code;

(d) a processor reading said real-time code if said real-time code bit indicates the presence of real-time code;

(e) storing said real-time code in one of said at least one of said plurality of data ways of said cache memory;

(f) locking said real-time code into said one of said at least one of said plurality of data ways of said cache memory to prevent overwrites; and

(g) executing said real-time code from said cache memory.

7. The method of claim **6**, wherein the step of locking said real-time code into said cache memory includes setting a lock bit associated with each line of cache memory containing real-time code.

8. The method of claim **6**, wherein said real-time code is unlocked from said cache memory after execution of the real-time code.

9. The method of claim **6**, wherein said executing said real-time code comprises said real-time code which operates on real-time data including multimedia data.

10. A cache system for storing and locking real-time code, comprising:

a first cache data way that is not lockable;

a second cache data way in which real-time code is lockable when stored therein;

a memory interface coupled to said first data way and said second data way;

a real-time address register that includes addresses associated with real-time code stored in said second cache data way; and

a comparator that compares an address received by said cache system with the contents of said real-time address register to determine if said address corresponds to real-time code.

11. The cache system of claim **10** wherein said real-time address register includes a starting address of said real-time code.

12. The cache system of claim **11** wherein said real-time address register further includes a size value indicating the size of the said real-time code.

13. The cache system of claim **12** wherein said real-time address register further includes a valid bit indicating whether said real-time code is to be locked in said second cache data way of said cache system.

14. The cache system of claim **13** further including:

a memory interface coupled to said first cache data way and said second cache data way; and

replacement logic coupled to said memory interface and said comparator.

15. The cache system of claim **14** wherein said comparator provides a signal to said replacement logic, wherein said signal indicates whether said address received by said cache system is an address included in said real-time address register.

16. The cache system of claim **15** wherein said valid bit from said real-time address register is provided to said replacement logic, wherein said replacement logic directs said memory interface to store real-time code exclusively into said second data way if said valid bit received from said real-time address register is set and said signal from said comparator indicates that an address received by said cache system is an address included in said real-time address register.

17. The cache system of claim **10**, wherein said real-time code operates on real-time data including multimedia data.

* * * * *

US005835908A

United States Patent [19]

Bennett et al.

[11] Patent Number: 5,835,908

[45] Date of Patent: Nov. 10, 1998

[54] PROCESSING MULTIPLE DATABASE TRANSACTIONS IN THE SAME PROCESS TO REDUCE PROCESS OVERHEAD AND REDUNDANT RETRIEVAL FROM DATABASE SERVERS

[75] Inventors: **John G. Bennett**, Bellevue; **Ketan Dalal**, Seattle, both of Wash.

[73] Assignee: **Microsoft Corporation**, Redmond, Wash.

[21] Appl. No.: **752,218**

[22] Filed: **Nov. 19, 1996**

[51] **Int. Cl.**^{6} **G06F 77/00**

[52] **U.S. Cl.** **707/10**; 711/122; 711/3; 711/136; 711/143; 395/800.01; 395/309; 707/1

[58] **Field of Search** 711/122, 3, 121, 711/143; 395/800.01, 309; 707/1, 10

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,023,776	6/1991	Gregor	711/122
5,241,641	8/1993	Iwasa et al.	711/122
5,261,066	11/1993	Jouppi et al.	711/122
5,276,848	1/1994	Gallagher et al.	711/121
5,307,477	4/1994	Taylor et al.	711/3
5,317,720	5/1994	Stamm et al.	711/143
5,386,547	1/1995	Jouppi	711/122
5,542,062	7/1996	Taylor et al.	711/3
5,584,013	12/1996	Cheong et al.	711/122
5,617,568	4/1997	Ault et al.	707/101
5,680,571	10/1997	Bauman	711/122
5,694,573	12/1997	Cheong et al.	711/122
5,717,919	2/1998	Kodavalla et al.	395/608
5,745,778	4/1998	Alfieri	395/800.01
5,774,682	6/1998	Benhase et al.	395/309

OTHER PUBLICATIONS

Leon Alkalaj and Rajendra Boppana, "Performance of Multi-Thread Execution in a Shared-Memory Multiprocessor", IEEE Jan. 1991, pp. 330–333.

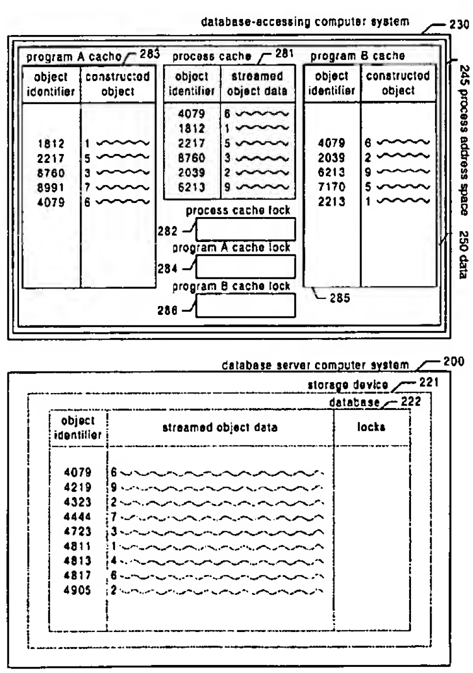
Radhika Thekkath and Susan Eggers, "Impact of Sharing-Based thread Placement on Multithreaded Archtectures", IEEE Jul. 1994, pp. 176–186.

Primary Examiner—Thomas G. Black
Assistant Examiner—Thuy Pardo
Attorney, Agent, or Firm—Seed and Berry LLP

[57] **ABSTRACT**

The present invention uses a segmented caching data structure to cache database objects provided by a database server. The database server provides database objects in response to requests by a number of different programs. The segmented caching data structure is made up of a single central cache and a number of program caches, each corresponding to one of the programs. When a database object is provided by the database server in response to a request by any of the programs, a copy of the database object is stored in the central cache. Another copy of the object is stored in the program cache for the program that requested the database object. When the segmented caching data structure is maintained in this manner, when a request is made by one of the programs a copy of the requested object stored in either of the central cache or the program cache for the program may be used, making it unnecessary for the database server to provide the requested database object.

16 Claims, 15 Drawing Sheets



PGPUB-DOCUMENT-NUMBER: 20020046326

PGPUB-FILING-TYPE: new

DOCUMENT-IDENTIFIER: US 20020046326 A1

TITLE: Management of caches in a data processing apparatus

PUBLICATION-DATE: April 18, 2002

US-CL-CURRENT: 711/128, 711/145 , 711/163

APPL-NO: 09/ 956238

DATE FILED: September 20, 2001

FOREIGN-APPL-PRIORITY-DATA:

COUNTRY	APPL-NO	DOC-ID	APPL-DATE
GB	0025468.0	2000GB-0025468.0	October 17, 2000

---------- **KWIC** ---------

Detail Description Paragraph - DETX (26):

[0073] Hence, in the preload and lock mode according to the preferred embodiment of the present invention only the desired data values will be locked into the locked way of the cache 30' provided the address range is selected

such that it does not include any address of the lockdown program.
Advantageously, this ensures that the data values of the lockdown program will
not occupy the lockdown way and prevent the correct operation of the lockdown
program. Hence, the lockdown program need not be specially written such that
it will usually operate in uncacheable memory as it will not get locked in the
cache 30', and because the lockdown program will not get locked in the cache
30' it is possible for the lockdown program to operate from the cache 30' and
achieve speed benefits.

US 20020046326A1

(19) **United States**

(12) **Patent Application Publication** (10) **Pub. No.: US 2002/0046326 A1**

Devereux (43) **Pub. Date: Apr. 18, 2002**

(54) **MANAGEMENT OF CACHES IN A DATA PROCESSING APPARATUS**

(76) Inventor: **Ian Victor Devereux**, Cambridge (GB)

Correspondence Address:
NIXON & VANDERHYE P.C.
8th Floor
1100 North Glebe Road
Arlington, VA 22201 (US)

(21) Appl. No.: **09/956,238**

(22) Filed: **Sep. 20, 2001**

(30) **Foreign Application Priority Data**

Oct. 17, 2000 (GB) 0025468.0

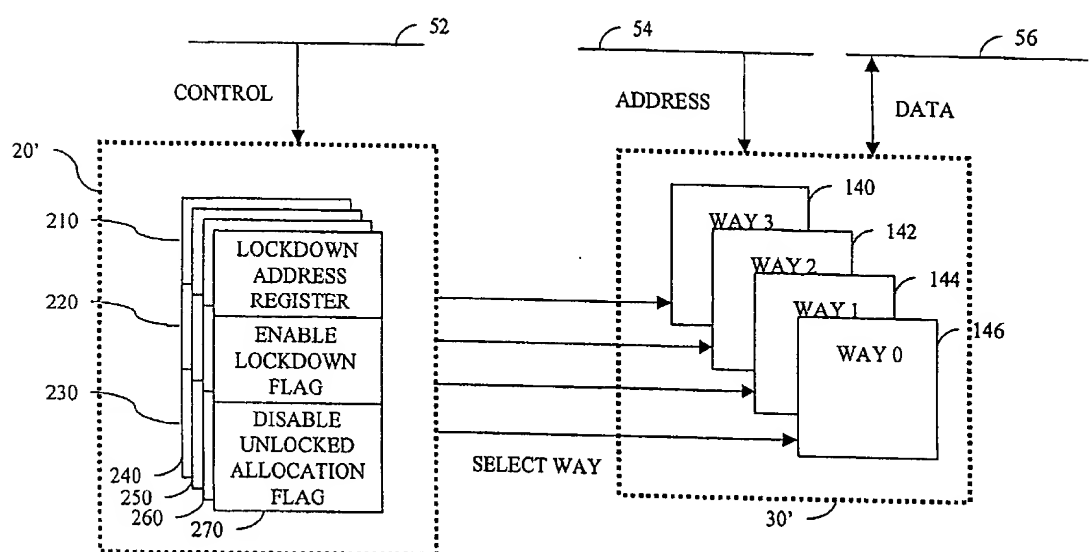
Publication Classification

(51) **Int. Cl.**[7] **G06F 12/08**

(52) **U.S. Cl.** **711/128**; 711/163; 711/145

(57) **ABSTRACT**

The present invention relates to the management of caches in a data processing apparatus, and in particular to the management of caches of the type where data in the cache may be designated as locked to prevent that data from being overwritten. The data processing apparatus comprises a processor, an n-way set associative cache having a plurality of entries, each entry being arranged to store one or more data values and a corresponding address identifier, the processor being operable to select one or more of the n-ways to operate in a lockdown mode, the lockdown mode being used to lock data values into the corresponding way, and a plurality of lockdown controllers. Each lockdown controller is associated with a corresponding way and comprises an address register arranged to store an address range specified by the processor such that, when the corresponding way is in the lockdown mode, only data values whose address identifiers are within the address range are locked into the corresponding way. This technique provides for reduced complexity during lockdown because in preferred embodiments a dedicated lockdown program is not required to carefully manage the storage of data values in the lockdown, the lockdown occurs automatically.



US-PAT-NO: 5694567

DOCUMENT-IDENTIFIER: US 5694567 A

TITLE: Direct-mapped cache with cache locking allowing expanded
contiguous memory storage by swapping one or more tag
bits with one or more index bits

DATE-ISSUED: December 2, 1997

US-CL-CURRENT: 711/3, 711/202

APPL-NO: 08/ 386025

DATE FILED: February 9, 1995

---------- KWIC ---------

Detailed Description Text - DETX (5):

The programmer can treat this embodiment as a cache RAM divided into 2 equal
portions, each portion servicing a contiguous half of the physical address
range. Thus, the programmer can store in the lower-order half of the physical
address range programs to be locked in the cache RAM, while storing other
programs contiguously in the upper-order of the physical address range, which

are serviced by the upper-order portion of the cache RAM. Thus, this embodiment realizes a "lockable" direct mapped cache without TLBs, additional sets of tag comparators, or additional page management or operating system software.

US-PAT-NO: 5913224

DOCUMENT-IDENTIFIER: US 5913224 A

TITLE: Programmable cache including a non-lockable data way and
a lockable data way configured to lock real-time data

DATE-ISSUED: June 15, 1999

US-CL-CURRENT: 711/125, 711/144 , 711/145 , 711/167

APPL-NO: 08/ 805554

DATE FILED: February 26, 1997

---------- KWIC ---------

Abstract Text - ABTX (1):
A computer system is disclosed which provides for execution of real-time
code from cache memory. A cache management unit provides the real-time code to
the cache memory from system memory upon a initiation of a read operation by a
processor. Once in cache memory, the processor executes the real-time code
from cache memory instead of system memory. The cache management unit detects
read hits to cache each time the processor requests an instruction of code that

is stored in the cache memory. Lock bits associated with each line of cache
lock the contents of the line preventing the line from being overwritten under
normal cache operation in which the least most recently used cached data is
replaced by presently accessed data. Alternatively, one of a plurality of
cache data ways may be dedicated to storing real-time code. Real-time code
stored in the dedicated data way is not replaceable and thus is locked.